

Sure 🤞 here's the explanation **without code** — short and clear:

---

## JDBC (Java Database Connectivity)

### **Definition:**

JDBC is a Java API that allows Java applications to connect to and interact with databases such as MySQL, Oracle, or PostgreSQL.

---

### **Purpose:**

It helps Java programs perform database operations like **inserting, updating, deleting, and retrieving** data in a standardized way.

---

### **Main Components:**

1. **DriverManager** – Manages database drivers and connections.
  2. **Connection** – Establishes a link between Java and the database.
  3. **Statement** – Sends SQL queries to the database.
  4. **ResultSet** – Stores the result returned from the database.
- 

### **Steps in JDBC Process:**

1. Load the database driver.
  2. Establish a connection.
  3. Create a statement.
  4. Execute SQL queries.
  5. Process the results.
  6. Close the connection.
- 

### **Advantages:**

- Platform-independent
- Supports multiple databases
- Provides a standard way to interact with databases

---

### Disadvantages:

- Requires manual coding for queries
  - Error handling can be complex
  - Not ideal for very large applications (use ORM frameworks instead)
- 

Would you like a **simple diagram explanation** of how JDBC connects Java to a database?

Good question, Nandhini 

Here's a clear explanation 

---

### Why JDBC is Used

JDBC is used because **Java programs need a way to interact with databases** — for storing, retrieving, and managing data.

---

### Main Reasons:

#### 1. Database Connectivity:

JDBC allows Java applications to connect to any relational database (like MySQL, Oracle, SQL Server).

#### 2. Standard Interface:

It provides a **common API** for all databases — so you don't need different code for each database.

#### 3. Data Operations:

Through JDBC, you can **insert, update, delete, or read** data using SQL queries directly from Java.

#### 4. Database Independence:

The same Java program can work with different databases by just changing the driver and connection URL.

#### 5. Integration with Java:

It integrates smoothly with Java EE technologies (like Servlets, JSP, Spring, Hibernate).

---

### In short:

JDBC acts as a **bridge between Java and databases**, allowing Java applications to **store and manage data efficiently**.

---

Would you like me to explain the **4 types of JDBC drivers** next? (It's an important interview question.)

Perfect, Nandhini  Let's understand **why JDBC is used** with a **real-time example** 

---

### Why JDBC? (With Real-Time Example)

Imagine you are developing an **E-commerce website** like *NanS Style* 

You need to:

- Store customer details (name, email, address)
  - Save product details (price, quantity, description)
  - Retrieve data when users log in or browse products
- 

### Without JDBC:

Your Java program cannot directly communicate with the database — it only understands Java, not SQL or database systems.

So, if you try to fetch or store data, it won't work.

---

### With JDBC:

JDBC acts as a **bridge** between your Java code and the database (like MySQL).

#### Example Flow in Real Life:

1. A customer signs up on your website.
2. The Java program collects the customer's info (name, email, password).
3. Using **JDBC**, the program connects to the database.
4. JDBC sends an **SQL query** (like `INSERT INTO users ...`) to save the customer data.
5. The database stores the details safely.

6. Later, when the customer logs in, JDBC helps **retrieve** that information using a SELECT query.
- 

### In Simple Terms:

JDBC helps Java applications **talk to the database** to store, update, or retrieve information — just like a translator between Java and SQL.

---

### Real-Life Analogy:

Think of JDBC as a **waiter** in a restaurant:

- You (Java program) tell the waiter what food (query) you want.
  - The waiter (JDBC) goes to the kitchen (database) to get it.
  - The waiter then brings the food (result) back to you.
- 

Would you like me to give a **real-world project scenario** (like in banking or online booking) showing how JDBC fits in?

Great question, Nandhini 

Here are the **prerequisites for learning and using JDBC** 

---

### Prerequisites for JDBC

To understand and work with JDBC effectively, you should have knowledge of the following:

---

### 1. Core Java Basics

You must know:

- Classes and Objects
- Methods and Constructors
- Exception Handling (try-catch)
- Packages and Import statements
- Interfaces and Inheritance

 Because JDBC is part of the **Java Standard Edition (Java SE)** and uses these core concepts.

---

## 2. Basic SQL Knowledge

You should understand how to use:

- **DDL commands:** CREATE, ALTER, DROP
- **DML commands:** INSERT, UPDATE, DELETE, SELECT
- **Constraints, Joins, and WHERE clauses**

 JDBC sends these SQL queries from Java to the database.

---

## 3. Database Setup

You need:

- A **Database Management System (DBMS)** like **MySQL**, **Oracle**, or **PostgreSQL** installed.
  - A basic understanding of how to create databases and tables.
- 

## 4. JDBC Driver

Each database needs a driver (like a connector) to communicate.

Example:

- For MySQL → mysql-connector-j.jar
- For Oracle → ojdbc8.jar

You must know how to **add the JDBC driver** to your Java project (in IDEs like IntelliJ or Eclipse).

---

## 5. Java IDE or Environment

You need:

- **JDK (Java Development Kit)** installed
  - An IDE like **IntelliJ IDEA**, **Eclipse**, or even **VS Code**
-

## 6. Basic Understanding of Client-Server Communication

Knowing how applications interact with servers and databases helps you understand how JDBC connects Java to backend databases.

---

### Summary Table

Prerequisite	Description
Core Java	Classes, objects, exception handling
SQL	Basic database commands
Database	MySQL/Oracle setup
JDBC Driver	Connector between Java and DB
IDE & JDK	Environment to run Java programs
Client-Server Basics	Understands data communication

---

Would you like me to give a **step-by-step setup guide** for connecting MySQL with Java using JDBC next?

Excellent, Nandini 

Let's learn clearly about the **JDBC Driver** — it's one of the most important concepts in JDBC.

---

### What is a JDBC Driver?

A **JDBC Driver** is a **software component (connector)** that enables your **Java program to communicate with a specific database**.

It acts like a **translator** between the **Java application** and the **database**.

---

### Example:

If your Java app uses MySQL, it doesn't directly understand how to talk to MySQL. So, you use a **MySQL JDBC Driver** (like mysql-connector-j.jar) to make that communication possible.

---

## Function of a JDBC Driver:

1. Establish a connection between Java and the database.
  2. Send SQL queries from Java to the database.
  3. Retrieve results from the database back to Java.
  4. Translate Java calls into database-specific calls.
- 

## ◆ Types of JDBC Drivers (4 Types)

Type	Name	Description	Example
1	JDBC-ODBC Bridge Driver	Converts JDBC calls into ODBC calls, then to the database.	Old Microsoft Access connections
2	Native API Driver	Uses native (C/C++) libraries of the database.	Oracle OCI Driver
3	Network Protocol Driver	Uses a middleware server to communicate between Java and the database.	IBM WebSphere Driver
4	Thin Driver (Pure Java)	Directly connects Java to the database using pure Java code — <b>most commonly used today.</b>	MySQL Connector/J, Oracle Thin Driver

---

## Most Commonly Used: Type 4 Driver

Because it is:

- 100% written in Java
  - Platform-independent
  - Easy to install and use
  - Doesn't need any external libraries or middleware
- 

## Example in Real Life:

Suppose you're building your **NanS Style e-commerce site** using Java + MySQL.  
You'll use the **MySQL Type 4 JDBC driver** (mysql-connector-j.jar) to:

- Connect your Java code to the MySQL database.
  - Send SQL queries (insert, update, delete, etc.)
  - Get results (like customer details or orders).
- 

### In Short:

A **JDBC driver** is a connector that lets Java programs **talk to databases** — and the **Type 4 driver** is the most popular and modern one.

---

Would you like me to explain **how to install and use the MySQL JDBC Driver** step by step in IntelliJ or Eclipse?

Perfect question, Nandhini 

Let's understand the **JDBC API** — it's the **core part** of how Java communicates with databases.

---

### What is JDBC API?

**JDBC API (Java Database Connectivity API)** is a set of **classes and interfaces** in the `java.sql` and `javax.sql` packages that help Java applications interact with databases.

It defines **how Java connects to a database, executes SQL queries, and processes results**.

---

### Purpose of JDBC API

It provides a **standard way** for all Java programs to:

- Connect to a database
  - Execute SQL statements
  - Retrieve, update, or delete data
  - Manage database transactions
- 

### Main Interfaces and Classes in JDBC API

Component	Description
<b>DriverManager</b>	Manages a list of database drivers and establishes connections.
<b>Driver</b>	Interface for database-specific driver implementation.
<b>Connection</b>	Represents a session/connection with a specific database.
<b>Statement</b>	Used to send SQL queries to the database.
<b>PreparedStatement</b>	Used for parameterized queries (safer and faster).
<b>CallableStatement</b>	Used to execute stored procedures in the database.
<b>ResultSet</b>	Represents the data returned from a query.
<b>SQLException</b>	Handles database-related errors and exceptions.

---

### Simple Understanding:

You can think of the **JDBC API** as a **toolbox** that contains all the tools (interfaces and classes) needed for:

1. Connecting to the database
  2. Sending SQL queries
  3. Getting results
  4. Closing the connection
- 

### Real-Life Example (Conceptual):

Imagine your **NanS Style e-commerce website**:

- You use the **Connection** object to connect to your product database.
- You use **Statement** to run SQL commands like `SELECT * FROM products`.
- The **ResultSet** gives you the product list from the database.
- Finally, you close the **Connection** to save resources.

All these actions are done using the **JDBC API**.

---

### In Short:

**JDBC API** provides all the **interfaces, classes, and methods** required to connect Java applications to databases and perform database operations.

---

Would you like me to explain each **JDBC interface (Connection, Statement, ResultSet, etc.)** one by one in simple terms next?

Perfect, Nandhini 

Let's understand **concrete classes in JDBC** with a **simple and clear analogy** 

---

### **Analogy: TV Remote and TV**

- The **JDBC Interface** is like a **remote control** — it defines buttons (like Power, Volume, Channel), but it doesn't actually show pictures or sound.
- The **Concrete Class (Driver Implementation)** is like the **actual TV** — it performs the real work when you press the buttons.

So, when you press “Power” on the remote (call a method on the interface), the **TV (driver's concrete class)** does the real action.

---

### **How it Relates to JDBC:**

<b>JDBC Concept</b>	<b>Analogy</b>
Connection, Statement, ResultSet (Interfaces)	Remote control – defines what actions can be done
JDBC Driver (Concrete Classes)	The actual TV – performs the real actions
Your Java Program	The person using the remote

---

### **Example:**

You write this in Java:

```
Connection con = DriverManager.getConnection(...);
```

It looks like you are using the **Connection** interface (remote), but internally, the MySQL Driver gives you a **real working object** like **ConnectionImpl** (the actual TV) that handles the connection.

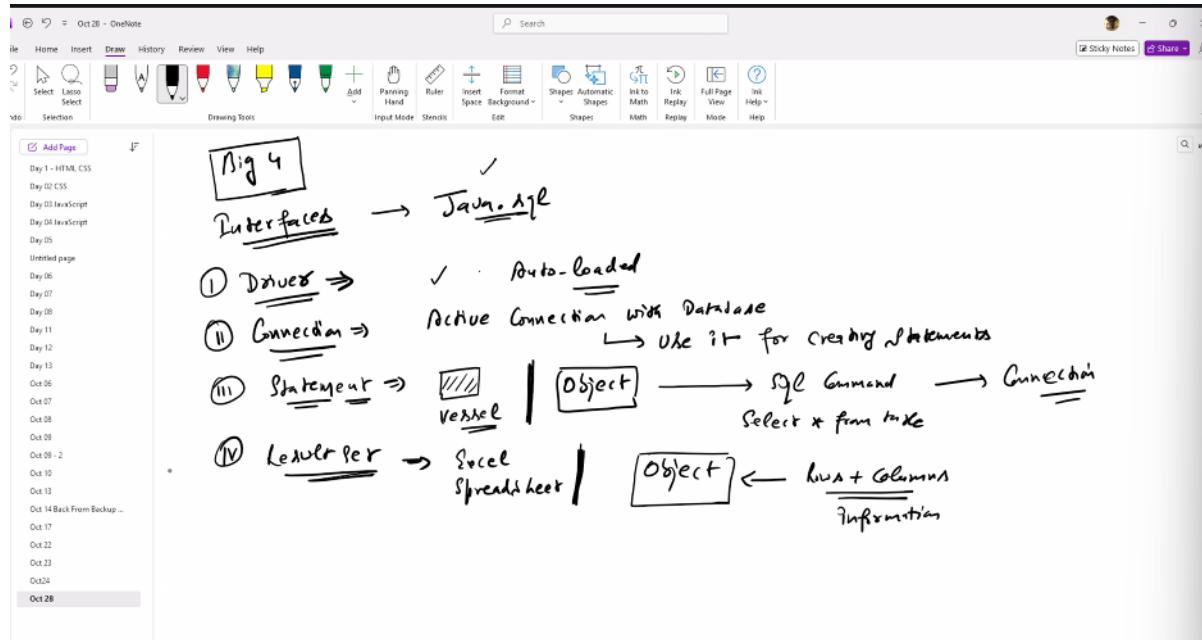
---

## In Short:

JDBC interfaces are **blueprints (remote)**, while driver's concrete classes are **real implementations (TV)** that do the actual work of connecting, querying, and fetching data.

---

Would you like me to give one more **real-life example** (like comparing it to a vehicle and driver) for better understanding?



The diagram illustrates the JDBC architecture as a Java API. It starts with 'Interfaces' leading to 'Java API'. The process is divided into four main steps:

- ① Driver → ✓ Auto-loaded
- ② Connection → Active Connection with Database  
↳ Use it for creating statements
- ③ Statement → SQL Command → Connection  
Select \* from table
- ④ Listener → Excel Spreadsheet | Object  
↳ Rows + Columns Information

Below this, there is a section labeled 'JDBC Workflow' with the following steps:

- ① Load JDBC Drivers
- ② Connection
- ③ Create Statements
- ④ Execute (SQL Queries)
- ⑤ Process ResultSet
- ⑥ Close Connection

On the right side of the page, there is a note 'My SQL'.

Perfect, Nandhini 💐 You've written the **JDBC Workflow** exactly right.

Let's clearly explain each step from your image 👏

---

### JDBC Workflow Explanation

The **JDBC workflow** represents the sequence of steps a Java program follows to connect to a database, execute SQL queries, and process results.

---

#### ① Load JDBC Driver

This step **loads the database driver** into memory.

The driver acts as a **bridge** between Java and the database.

##### Example:

For MySQL, the driver is `com.mysql.cj.jdbc.Driver`.

It helps Java recognize how to communicate with MySQL.

Think of it as installing the translator before starting the conversation.

---

#### ② Establish Connection

Once the driver is loaded, you must **connect to the database** using:

- **URL** (database location)
- **Username**
- **Password**

This step creates a **session** between your Java program and the database.

Example URL: `jdbc:mysql://localhost:3306/studentdb`

Analogy: Like making a phone call to the database — now both sides are connected.

---

#### ③ Create Statement

After the connection, you need a **Statement object** to send SQL queries.

You can use:

- `Statement` → for simple queries
- `PreparedStatement` → for queries with parameters

- `CallableStatement` → for stored procedures

Analogy: Writing your question before asking it over the phone.

---

#### ④ Execute (SQL Queries)

Now, use the statement to **run your SQL query** (e.g., SELECT, INSERT, UPDATE, DELETE).

- `executeQuery()` → for SELECT
- `executeUpdate()` → for INSERT, UPDATE, DELETE

This is where the actual action happens inside the database.

---

#### ⑤ Process ResultSet

If your query returns data (like `SELECT * FROM students`),  
the results come back inside a **ResultSet object**.

You can loop through it to read each row and column.

Analogy: Reading the answer the database sent back.

---

#### ⑥ Close Connection

Finally, **close all connections** (`ResultSet`, `Statement`, `Connection`) to free up resources.  
Leaving them open can cause **connection leaks** or **overflow errors**.

Analogy: Hanging up the phone after finishing your conversation.

---

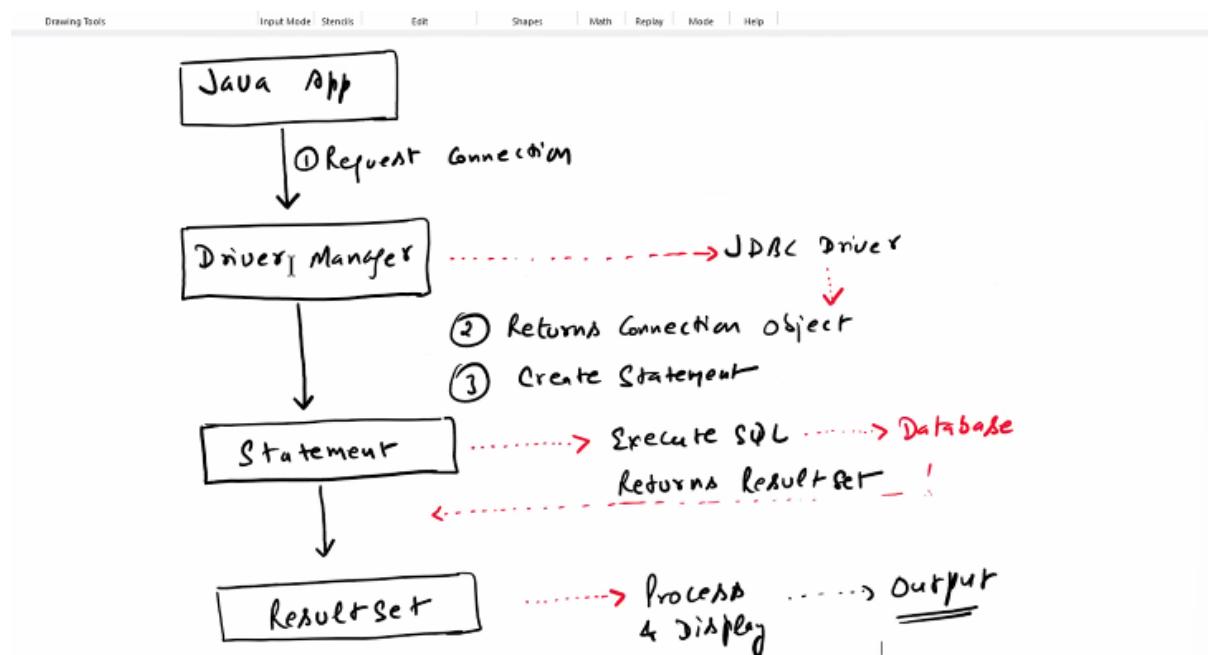
#### In Short:

Step	Action	Purpose
1	Load JDBC Driver	Initialize database driver
2	Establish Connection	Connect Java with DB
3	Create Statement	Prepare SQL command
4	Execute SQL Query	Run the command in DB
5	Process ResultSet	Read and use data

Step	Action	Purpose
6	Close Connection	Release resources

---

Would you like me to create a **simple diagram or flowchart** of this JDBC workflow for your notes or presentation?



Excellent, Nandhini 🌸

Let's understand clearly what `java.sql.*;` means in JDBC 👇

---

### 💡 Meaning of `java.sql.*;`:

In Java,

`import java.sql.*;`

means you are **importing all the classes and interfaces** from the **java.sql package**, which are needed to work with **JDBC (Java Database Connectivity)**.

---

### ⚙️ Why It's Used

JDBC provides several built-in interfaces and classes to:

- Connect Java with databases
- Execute SQL queries

- Process results

All these are defined inside the **java.sql** package.

---

### Important Interfaces & Classes Inside **java.sql**

Component	Type	Description
DriverManager	Class	Manages database drivers and connections.
Connection	Interface	Represents a connection/session with the database.
Statement	Interface	Used to send SQL queries to the database.
PreparedStatement	Interface	Used for precompiled and parameterized SQL queries.
CallableStatement	Interface	Used to call stored procedures.
ResultSet	Interface	Stores and processes the results of queries.
SQLException	Class	Handles database access errors and exceptions.
Date, Time, Timestamp	Classes	Represent SQL date/time values.

---

### Example Use:

When you write:

```
import java.sql.*;
```

you can directly use:

```
Connection con;
```

```
Statement stmt;
```

```
ResultSet rs;
```

without writing:

```
import java.sql.Connection;
```

```
import java.sql.Statement;
```

```
import java.sql.ResultSet;
```

every time.

---

### In Simple Words:

`java.sql.*;` imports everything you need for **JDBC programming** — like **Connection**, **Statement**, **ResultSet**, and **DriverManager** — to make your Java program communicate with a database.

---

Would you like me to explain the **difference between java.sql and javax.sql** (it's an important interview question)?

Perfect — this diagram you shared clearly shows the **JDBC (Java Database Connectivity) process flow**. Let's explain it step by step 

---

## JDBC Workflow Explanation

### 1. Java Application

- Your Java program wants to connect to a database (like MySQL, Oracle, etc.).
  - It sends a **request for a connection**.
- 

### 2. DriverManager

- The **DriverManager** class manages all JDBC drivers.
  - It loads the **JDBC Driver** (like `com.mysql.cj.jdbc.Driver`).
  - After loading the driver, it:
    1. **Returns a Connection object** → establishes a connection to the database.
    2. **Creates a Statement object** → allows you to write and execute SQL queries.
- 

### 3. Statement

- The **Statement** object is used to send SQL queries to the database.
  - Example:
  - `Statement stmt = con.createStatement();`

- o `ResultSet rs = stmt.executeQuery("SELECT * FROM students");`
  - It **executes the SQL command** and sends it to the **database**.
  - Then, it **returns a ResultSet** (the result of the query).
- 

#### 4. ResultSet

- The **ResultSet** object holds the data retrieved from the database.
  - You can process it row by row.
  - `while (rs.next()) {`
  - `System.out.println(rs.getString("name"));`
  - `}`
  - Finally, the data is **processed and displayed as output**.
- 

#### Flow Summary

Java App

↓

DriverManager

↓ (connects using JDBC Driver)

Statement

↓ (executes SQL)

ResultSet

↓ (processes result)

Output

---

#### Important Package

`import java.sql.*;`  
→ This imports all the JDBC classes like Connection, Statement, ResultSet, DriverManager.

---

Would you like me to give a **small sample code** showing this full JDBC flow (connect → query → display)?

Step	Component	Action	Result
1	Java App	Requests a connection	Sends to DriverManager
2	DriverManager	Loads JDBC driver	Returns Connection object
3	Connection/Statement	Executes SQL query	Sends to Database
4	Database	Processes SQL	Returns ResultSet
5	ResultSet	Holds query output	Processed & displayed

Here's your table in a **well-formatted, professional version** 👇

---

### JDBC Workflow Summary

Step	Component	Action	Result
1	Java Application	Requests a connection	Sends request to DriverManager
2	DriverManager	Loads the JDBC driver	Returns a Connection object
3	Connection / Statement	Creates Statement and executes SQL query	Sends query to the Database
4	Database	Processes the SQL command	Returns a ResultSet object
5	ResultSet	Holds the query output	Processed and displayed as output

---

Would you like me to include the **corresponding Java code snippet** for these 5 steps next?

The screenshot shows a web browser window with three tabs: 'Teamware', 'Post Attenc...', and 'JDBC work...'. The active tab is 'chatgpt.com/c/690049d0-febc-8320-af8e-6ec1e75...'. The page content is as follows:

ChatGPT

```
}
```

Copy code

- The data is then processed and displayed as output.

---

### Overall JDBC Workflow Summary

Step	Component	Action	Result
1	Java App	Requests a connection	Sends to Driver
2	DriverManager	Loads JDBC driver	Returns Connection object
3	Connection/Statement	Executes SQL query	Sends to Database
4	Database	Processes SQL	Returns ResultSet
5	ResultSet	Holds query output	Processed & displayed

Ask anything

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

Sure 🤝 Here's the explanation of **types of JDBC drivers** in simple **paragraph (non-tabular)** form:

## Types of JDBC Drivers

### 1. Type 1: JDBC–ODBC Bridge Driver

This driver uses the ODBC (Open Database Connectivity) driver to connect Java applications to databases. It converts JDBC method calls into ODBC function calls. It's easy to use but slow and platform-dependent because it requires ODBC to be installed on the client system.

👉 Example: `sun.jdbc.odbc.JdbcOdbcDriver` (now obsolete in newer Java versions).

---

### 2. Type 2: Native API (Partly Java) Driver

This driver converts JDBC calls into native database API calls. It uses database-specific client libraries (for example, Oracle's OCI). It performs better than Type 1 but is not portable because it depends on native database libraries that must be installed on every client machine.

👉 Example: Oracle's `oracle.jdbc.driver.OracleDriver`.

---

### 3. Type 3: Network Protocol (Middleware) Driver

This driver uses a middle-tier (middleware) server between the Java application and the database. JDBC calls are sent to the middleware server, which then translates them into database-specific calls. It is database-independent and doesn't need client libraries, but it's slower due to the extra network layer.

👉 Example: WebLogic or IDS Server JDBC driver.

---

### 4. Type 4: Thin Driver (Pure Java Driver)

This is a pure Java driver that converts JDBC calls directly into the database's native protocol. It doesn't require any additional software or libraries and is platform-independent. It provides the best performance and is the most commonly used driver today.

👉 Example: MySQL's `com.mysql.cj.jdbc.Driver`, Oracle's `oracle.jdbc.OracleDriver`.

---

#### ✓ In short:

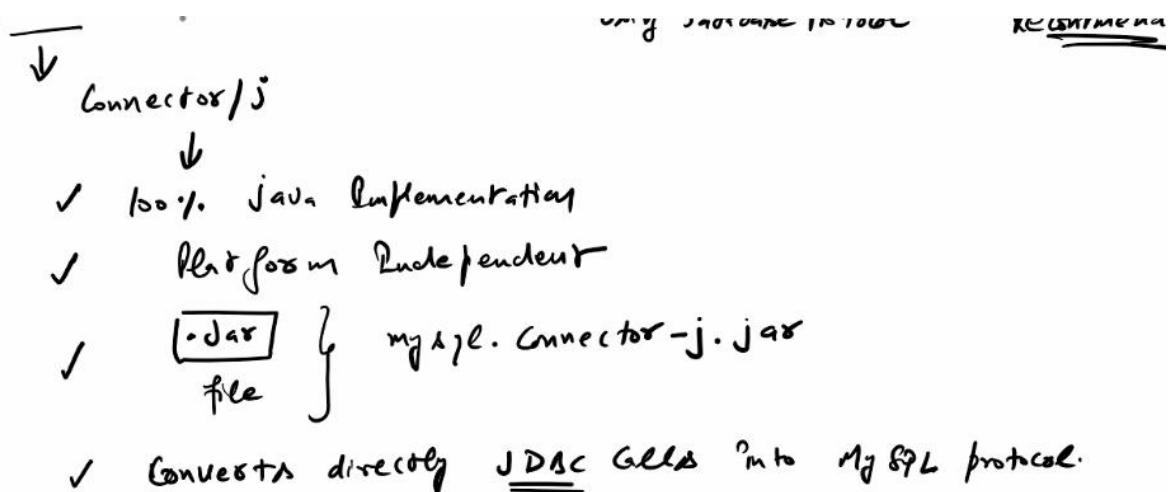
- Type 1 → Bridge (Old, slow, not portable)
- Type 2 → Native API (Fast but not portable)
- Type 3 → Middleware (Flexible but slower)

- Type 4 → Thin Driver (Fast, portable, widely used)
- 

Would you like me to add a **real-world analogy** (like comparing them to communication methods — interpreter, translator, etc.) to make it easier to remember?

“**Not portable**” means that the driver **cannot run on all systems or platforms** — it depends on some **specific software or libraries** that must be installed on each computer where your Java program runs.

Drawing Tools			
	Input Mode	Shapes	Math
<u>Type</u>			
Type 1	JDBC-ODBC Bridge	→ Very old	Deprecated
Type 2	Native API Drivers	use Native DB Client Libraries	Not Portable
Type 3	N/w Protocol Drivers	use Middleware Servers	Complex Setup
Type 4	pure Java Drivers	Communicates Directly using JDBC API	✓ Modern Recommended
	↓		
	Connector/J		



Perfect 🌟 Let's clearly explain what's shown in your note about **Connector/J**:

---

## MySQL Connector/J Explanation

**Connector/J** is the **JDBC driver** used by Java applications to connect to **MySQL databases**.

It's an example of a **Type 4 (Thin) Driver**.

---

## Key Points from Your Note

### 1. 100% Java Implementation

- The driver is written entirely in Java — no native code.
- Because of this, it's **platform-independent** (runs on Windows, Linux, macOS, etc.).

### 2. Platform Independent

- You don't need any external software or native libraries.
- The same .jar file works on all systems that have Java installed.

### 3. .jar File

- The driver comes as a **JAR file** named  mysql-connector-j.jar (sometimes versioned like mysql-connector-j-8.0.xx.jar).
- You add this JAR to your project's classpath so Java can use it to connect to MySQL.

### 4. Converts JDBC Calls Directly into MySQL Protocol

- When you execute a SQL query in Java (like SELECT \* FROM students;), the driver **translates that JDBC command** into a **MySQL-specific protocol** (the language MySQL understands).
  - This means no middle layer — your Java app talks **directly** to the MySQL database.
- 

## In Short

### **Connector/J = Type 4 JDBC Driver for MySQL**

It is a pure Java .jar file that directly converts JDBC calls into MySQL commands, making it fast, portable, and easy to use.

---

Would you like me to show the **exact steps to add mysql-connector-j.jar** in IntelliJ or Eclipse for a JDBC project?

Great question, Nandhini 

Let's break it down simply 

---

### Why Java is Platform Independent

Java is **platform independent** because **compiled Java programs (bytecode)** can run on **any operating system** that has a **Java Virtual Machine (JVM)**.

---

### Step-by-Step Explanation

1. **You write Java code** → Example: Hello.java
  2. **The Java compiler (javac)** converts it into **bytecode** → Hello.class
    - This bytecode is **not specific** to Windows, macOS, or Linux.
  3. **The JVM** (Java Virtual Machine) on each platform **interprets and runs** the same bytecode.
    - JVMs are **platform-specific**, but bytecode is **platform-independent**.
- 

### Simple Analogy

Think of Java like a movie DVD  :

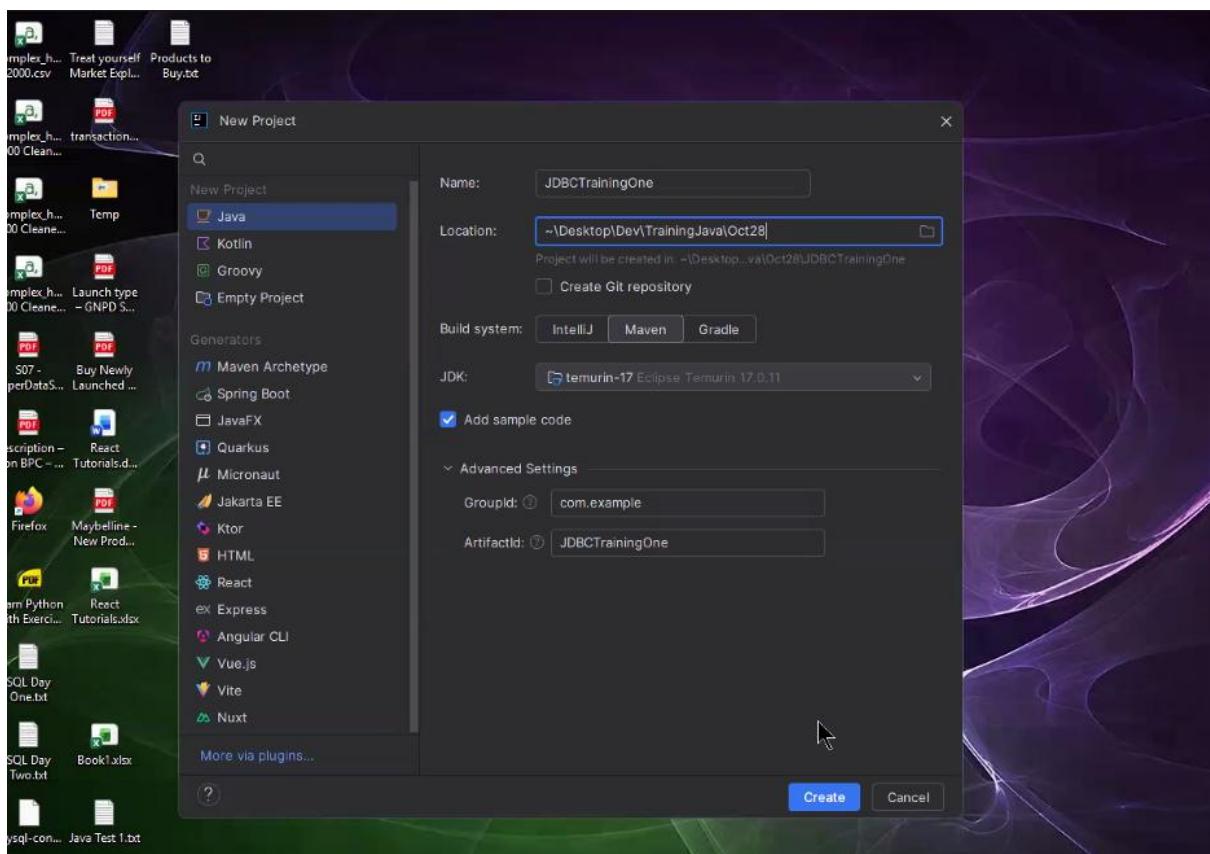
- The **movie (bytecode)** is the same everywhere.
  - The **DVD player (JVM)** may differ — one for TV, one for computer — but all can play the same movie.
- 

### In Short

**Java is platform independent** because its compiled code (bytecode) runs on any system that has a JVM, regardless of the underlying hardware or operating system.

---

Would you like me to add a small diagram showing how Java → Bytecode → JVM → Any OS works?



The screenshot shows the IntelliJ IDEA interface with the project 'JDBCTraining' open. The left sidebar displays the project structure with files like .idea, .mvn, src, main, test, .gitignore, pom.xml, and external Libraries. The right pane shows the content of the pom.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>JDBCTraining</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>25</maven.compiler.source>
        <maven.compiler.target>25</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

The screenshot shows the IntelliJ IDEA interface with the project 'JDBCTrainingOne' open. The left sidebar displays the project structure with files like .idea, .mvn, src, main, test, .gitignore, pom.xml, and External Libraries. The right pane shows the content of the pom.xml file, which includes a dependency for MySQL Connector-J:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.learn.jdbc</groupId>
    <artifactId>JDBCTrainingOne</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>8.3.0</version>
        </dependency>
    </dependencies>
</project>
```

## configuration

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.4.0</version> <!-- latest stable version as of 2025 -->
```

```

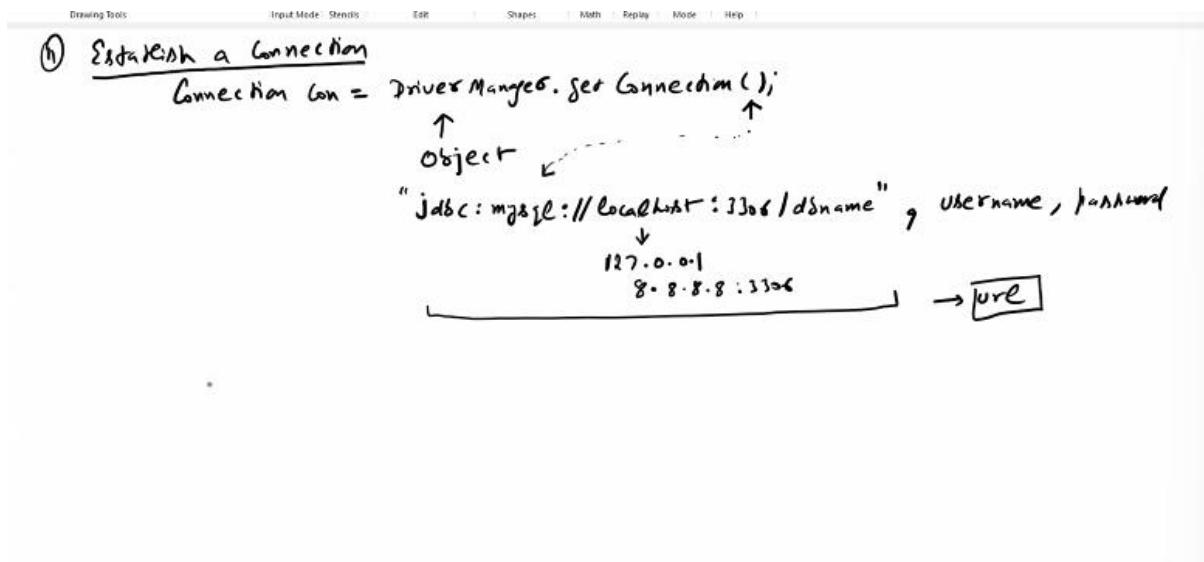
</dependency>
</dependencies>

```

After save and reload maven symbol

The screenshot shows the IntelliJ IDEA interface with the Maven tool window open. The Maven window displays the following structure:

- Maven** node
  - JDBCTraining** node
    - Lifecycle**
    - Plugins**
      - clean
      - compiler
      - deploy
      - install
      - jar
      - resources
      - site
      - surefire
    - Dependencies**
      - com.mysql:mysql-connector
      - com.google.protobuf:protobuf-java
    - Repositories**



Excellent question, Nandini ✨

In **JDBC**, there are **three main types of statements**, each used for different types of SQL execution and use cases.

## ✿ Types of Statements in JDBC

Type Interface/Class	Use Case	Example Query Type
1 Statement	For executing <b>static SQL queries</b> (queries that don't change).	SELECT * FROM students;
2 PreparedStatement	For executing <b>parameterized/dynamic queries</b> . Prevents SQL Injection.	SELECT * FROM students WHERE id = ?;
3 CallableStatement	For executing <b>stored procedures</b> in the database.	CALL getStudentDetails(?);

---

#### ◆ 1. Statement

Used when SQL query is fixed and does not change at runtime.

```
Statement stmt = con.createStatement();
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

 **Best for:** Simple queries (DDL or DML)

 **Not secure** if user input is added directly (vulnerable to SQL Injection)

---

#### ◆ 2. PreparedStatement

Used when the query has **parameters (?)** that change dynamically.

```
String sql = "SELECT * FROM students WHERE id = ?";
```

```
PreparedStatement pstmt = con.prepareStatement(sql);
```

```
pstmt.setInt(1, 101);
```

```
ResultSet rs = pstmt.executeQuery();
```

 **Advantages:**

- Prevents **SQL Injection**
- Executes faster (precompiled)
- Cleaner and safer code

 **Best for:** Repeated or dynamic queries (like login forms)

---

### ◆ 3. CallableStatement

Used to **call stored procedures** from the database.

```
CallableStatement cstmt = con.prepareCall("{call getStudentDetails(?)}");  
cstmt.setInt(1, 101);  
  
ResultSet rs = cstmt.executeQuery();
```

#### Advantages:

- Executes precompiled SQL stored in the database
- Better performance for complex logic
- Supports **IN, OUT, and INOUT** parameters

#### Best for: Business logic stored in the database (e.g., procedures, functions)

---

#### Quick Comparison

Feature	Statement	PreparedStatement	CallableStatement
SQL type	Static	Dynamic	Stored Procedure
Parameters	No	Yes (?)	Yes (IN/OUT)
SQL Injection	Possible	Prevented	Prevented
Performance	Slow	Faster (precompiled)	Fastest
Use Case	Simple fixed queries	Repeated queries with input	Complex database operations

---

#### Summary

- Use **Statement** for one-time, fixed queries.
  - Use **PreparedStatement** for user inputs and dynamic queries.
  - Use **CallableStatement** for stored procedures.
- 

Would you like me to show a **program that uses all three types in one file** for better understanding?

Perfect, Nandhini 🌸 I can see your handwritten JDBC notes — let's neatly convert **all the content** into clean, well-formatted text with small explanations for each part.

---



## MySQL – In order to work with JDBC

You need to import the following classes and interfaces from the **java.sql** package:

```
import java.sql.DriverManager;  
  
import java.sql.Connection;  
  
import java.sql.Statement;  
  
import java.sql.PreparedStatement;  
  
import java.sql.CallableStatement;  
  
import java.sql.ResultSet;  
  
import java.sql.ResultSetMetaData;  
  
import java.sql.DatabaseMetaData;  
  
import java.sql.SQLException;
```

---



## Steps in JDBC

### Step 1: Load the Driver

Before connecting to the database, you must **load the JDBC driver**.

Two methods can be used:

#### Method 1 – Using Class.forName()

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- This method loads the MySQL JDBC driver class into memory.
- It registers the driver automatically with DriverManager.

#### Method 2 – Using DriverManager.registerDriver()

```
DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
```

- This explicitly registers the driver object with the DriverManager.
- 

### Step 2: Establish a Connection

After loading the driver, we connect Java to the database.

```
Connection con = DriverManager.getConnection(  
    "jdbc:mysql://localhost:3306/databasename", "root", "password");
```

- **DriverManager.getConnection()** → Used to create a connection.
- It returns a **Connection** object (con), which represents the link between Java and MySQL.

◆ Example:

```
String url = "jdbc:mysql://localhost:3306/learningmysql";  
  
String user = "root";  
  
String pass = "12345";  
  
Connection con = DriverManager.getConnection(url, user, pass);
```

---

### Step 3: Fire (Execute) SQL Queries

Once the connection is established, we can send SQL commands.

We can create SQL queries using three types of statement objects:

Type	Interface	Use Case
1	Statement	To execute simple (static) SQL queries
2	PreparedStatement	To execute parameterized queries
3	CallableStatement	To execute stored procedures

Example:

```
Statement stmt = con.createStatement();  
  
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

or using PreparedStatement:

```
PreparedStatement pstmt = con.prepareStatement(  
    "INSERT INTO students (id, name) VALUES (?, ?)");  
  
pstmt.setInt(1, 101);  
  
pstmt.setString(2, "Nandhini");
```

```
pstmt.executeUpdate();
```

or using CallableStatement (for stored procedures):

```
CallableStatement cstmt = con.prepareCall("{call getStudentDetails(?)}");
```

```
cstmt.setInt(1, 101);
```

```
cstmt.execute();
```

---

#### Step 4: Process the Results

If the SQL query returns data (like a SELECT statement), you use a **ResultSet** to read it.

```
while (rs.next()) {  
    System.out.println(rs.getInt("id") + " " + rs.getString("name"));  
}
```

---

#### Step 5: Close the Connection

It's important to close all database connections after use.

```
con.close();  
stmt.close();
```

This frees up memory and prevents database locking issues.

---

#### Summary of JDBC Steps

Step	Description	Example
1 Load Driver	Load JDBC driver class	Class.forName("com.mysql.cj.jdbc.Driver");
2 Establish Connection	Connect to database	DriverManager.getConnection(url, user, pass);
3 Create Statement	Prepare SQL query	Statement stmt = con.createStatement();
4 Execute Query	Run query	stmt.executeQuery("SELECT * FROM table");

Step	Description	Example
5 Process Results	Read data	rs.next()
6 Close	Close connection	con.close();

Would you like me to extend this with a **simple complete example program** showing all these 5 steps together (for your notes)?

III fire SQL Query  
✓ Create a Query → Query Statement or Prepared Statement or Callable Statement → Procedures

String query = "Select \* from sales";  
Statement stmt = Conn.createStatement();  
ResultSet resultSet = stmt.executeQuery(query);  
return resultSet;

IV Process the Data  
✓ Iterate through ResultSet  
while (resultSet.next()) {  
≡

V Con.close()

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** Shows "JDBCTraining" as the active project.
- File Bar:** Shows "Main.java" as the active file.
- Code Editor:** Displays the Java code for "Main.java". The code connects to a MySQL database using JDBC. It includes imports for java.sql.Connection, java.sql.DriverManager, and java.sql.SQLException. The main method establishes a connection and prints "Connected to the database!" to the console.
- Run Tab:** Shows the run configuration "Main".
- Output Tab:** Shows the console output:

```
"C:\Program Files\Java\jdk-25+36\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2025.2.2\lib\idea Performance
Connecting to the database...
Connected to the database!
Connection closed.

Process finished with exit code 0
```

```
// Package declaration - this defines the package name for your class
package com.example;

// Importing the necessary JDBC classes
import java.sql.Connection; // Used to create a connection with the database
import java.sql.DriverManager; // Used to manage JDBC drivers and establish connection
import java.sql.SQLException; // Used to handle SQL-related exceptions

public class Main {
    public static void main(String[] args) {
        // Database credentials (username and password)
        String user = "root"; // MySQL username
        String pass = "12345"; // MySQL password
```

```
// URL to connect to the MySQL database
// jdbc:mysql:// - protocol for MySQL database
// localhost - means the database is running on your local computer
// 3306 - default MySQL port number
// learningmysql - name of the database you created in MySQL
String url = "jdbc:mysql://localhost:3306/learningmysql";

System.out.println("Connecting to the database...");

// Step 1: Declare the connection variable
// Initially set to null (no connection yet)
Connection conn = null;

try {
    // Step 2: Load the MySQL JDBC Driver
    // This tells Java which database driver to use.
    // It's optional for modern Java versions but good practice for beginners.
    Class.forName("com.mysql.cj.jdbc.Driver");

    // Step 3: Establish connection to the database
    // DriverManager tries to connect using the given URL, username, and password.
    conn = DriverManager.getConnection(url, user, pass);

    // If connection succeeds, this line executes
    System.out.println(" ✅ Connected to the database!");

} catch (Exception e) {
    // Step 4: If any error occurs (e.g., wrong password or driver issue)
```

```
// the catch block will execute  
  
System.out.println("✖ Error: " + e.getMessage());  
  
} finally {  
  
    // Step 5: This block runs no matter what happens (success or failure)  
  
    // Used to safely close the connection to avoid memory leaks  
  
    if (conn != null) {  
  
        try {  
  
            conn.close() // Close the connection when done  
  
            System.out.println("🔒 Connection closed.");  
  
        } catch (SQLException e) {  
  
            // Handle any errors that occur while closing  
  
            System.out.println("⚠ Error while closing: " + e.getMessage());  
  
        }  
  
    }  
  
}  
  
}
```

---

```
package com.example;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class Main {  
    public static void main(String[] args) {  
        String user = "root";  
        String pass = "12345";  
        String url = "jdbc:mysql://localhost:3306/learningmysql";  
  
        System.out.println("Connecting to the database...");  
        Connection conn = null;
```

```

try{
    // Establish connection
    conn = DriverManager.getConnection(url, user, pass);
    System.out.println(" Connected to the database!");

}

catch (Exception e){
    System.out.println(" Error: " + e.getMessage());
}

finally
{
    // Close connection safely
    if (conn != null){
        try{
            conn.close();
            System.out.println(" Connection closed.");
        } catch (SQLException e){
            System.out.println(" Error while closing: " + e.getMessage());
        }
    }
}
}

```

ideal way

2 nd way// another method—modern way

```

package com.example;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Main {
    public static void main(String[] args) {
        String user = "root";
        String pass = "12345";
        String url = "jdbc:mysql://localhost:3306/learningmysql"; // ✓ Correct URL

        System.out.println("Connecting to the database...");
    }
}

```

```
try( Connection conn = DriverManager.getConnection(url, user, pass)) {  
  
    // Establish connection  
    if(conn != null){  
        System.out.println(" Connected to the database!");  
    }  
} catch (Exception e){  
    System.out.println("Connection Failed... ");  
}  
}  
}
```

Excellent question, Nandhini 🌸 — both programs connect to the database correctly, but they differ in **how they manage the database connection**.

Let's clearly compare them 👇

---

### ✳️ 1 First Way — Traditional (Old / Manual Closing Method)

**Code Style:**

```
Connection conn = null;  
  
try{  
  
    conn = DriverManager.getConnection(url, user, pass);  
  
    System.out.println("Connected!");  
  
} catch (Exception e){  
  
    System.out.println("Error: " + e.getMessage());  
  
} finally{  
  
    if (conn != null){  
  
        conn.close(); // manually closing connection  
  
    }  
  
}
```

### 🔍 Explanation:

- Here, we **create the connection manually** and **close it manually** inside finally.

- You need to **explicitly check if conn != null** and handle exceptions while closing.
- If you **forget to close**, the connection **remains open**, causing **memory leaks** or **database connection exhaustion**.

### Drawbacks:

- More code.
  - Easy to forget to close resources.
  - Can cause **resource leakage** if exception occurs before closing.
  - Harder to read and maintain.
- 

## 2 Second Way — Modern (Try-with-Resources Method)

### Code Style:

```
try (Connection conn = DriverManager.getConnection(url, user, pass)) {  
    System.out.println("Connected!");  
}  
catch (Exception e) {  
    System.out.println("Connection Failed...");  
}
```

### Explanation:

- Uses **Java 7+ feature: Try-with-Resources**.
- Automatically **closes the connection** after the try block, even if an exception occurs.
- Cleaner, safer, and more readable.
- No need for a finally block or manual close.

### Advantages:

- No chance of forgetting to close.
  - Reduces boilerplate code.
  - Automatically calls conn.close() internally.
  - Ideal for modern Java JDBC programs.
-

## Summary Table

Feature	Old Way (Manual Close)	Modern Way (Try-with-Resources)
Introduced	Before Java 7	Java 7 and later
Closing Method	Manual using finally	Automatic after try block
Code Length	Longer, with extra checks	Short and clean
Error Risk	Possible resource leaks	Automatically managed
Best Practice	Outdated	 Recommended

---

### In short:

-  The second (modern) way using **try-with-resources** is the **ideal approach** because it's **clean, safe, and efficient**.
- 

Would you like me to show how both methods behave when an **exception occurs before closing** (so you can see why the modern one is safer)?

### 3rd way

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Main {
    public static void main(String[] args) {
        String user = "root";
        String pass = "12345564";
        String url = "jdbc:mysql://localhost:3306/learningmysql"; //  Correct URL

        System.out.println("Connecting to the database...");

        try {
            Connection conn = DriverManager.getConnection(url, user, pass);
            // Establish connection
            if(conn != null) {
                System.out.println(" Connected to the database!");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```

        }
    } catch (Exception e) {
        System.out.println("Connection Failed... ");
    }
}
}

```

```

package com.learn.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.
public class Main {
    public static void main(String[] args) {
        String user = "root";
        String password = "123455445";
        String url = "jdbc:mysql://localhost:3306/learningmysql";

        System.out.println("Connecting to the database....");

        try{
            Connection conn = DriverManager.getConnection(url, user, password);
            if(conn != null){
                System.out.println("Connected to the database... ");
            }
        }
        catch(Exception e){
            System.out.println("Connection Failed... ");
        }
    }
}

```

Perfect, Nandhini 💐 — let's go through your code **line by line** so you fully understand what's happening in this JDBC program.

### Program Explanation

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

```

- ✓ These are **JDBC classes** from the `java.sql` package:

- **Connection** → represents a link between Java and your database.
- **DriverManager** → manages JDBC drivers and establishes database connections.

- **SQLException** → handles SQL-related errors (though you used general Exception here).
- 

```
public class Main {  
    public static void main(String[] args) {
```

This is the **entry point** of the program.

---

```
String user = "root";  
String pass = "12345564";  
String url = "jdbc:mysql://localhost:3306/learningmysql";
```

- user: MySQL username.
- pass: Password for the MySQL account.
- url: **JDBC URL** that tells Java how to connect to the database.
  - jdbc:mysql:// → JDBC protocol for MySQL.
  - localhost → Database is on your local computer.
  - 3306 → Default MySQL port.
  - /learningmysql → The name of your database.

 **Example meaning:**

“Connect to the database named *learningmysql* on localhost at port 3306 using user *root* and password *12345564*.”

---

```
System.out.println("Connecting to the database...");
```

Displays a message before attempting the connection.

---

```
try {  
    Connection conn = DriverManager.getConnection(url, user, pass);  
    • The DriverManager.getConnection() method tries to:  
        1. Find a suitable JDBC driver (e.g., com.mysql.cj.jdbc.Driver).
```

2. Use your credentials and URL to open a connection.
  3. Return a **Connection object** (conn) if successful.
- 

```
if(conn != null){  
    System.out.println(" Connected to the database!");  
}
```

- Checks if the connection object is not null — which means the connection was successful.
- 

```
} catch (Exception e){  
    System.out.println("Connection Failed... ");  
}
```

- If something goes wrong (e.g., wrong password, database not running, missing driver), it will:

- Jump to the **catch block**
  - Print “Connection Failed...”
- 

## How it works in real time

1. You start your MySQL server.
  2. The Java program runs and uses JDBC to talk to MySQL.
  3. If username, password, and database are correct →  “Connected to the database!”
  4. If any step fails →  “Connection Failed...”
- 

## Notes / Improvements

- You’re not **closing the connection**, which can cause resource leaks.  
→ You can add:
  - conn.close();

after the success message.

- Or even better, use **try-with-resources**:
  - ```
try (Connection conn = DriverManager.getConnection(url, user, pass)) {
```
  - ```
    if (conn != null) System.out.println("Connected!");
```
  - ```
}
```
- 

## In Short

| Concept | Meaning |
|---------|---------|
|---------|---------|

|                      |                                                |
|----------------------|------------------------------------------------|
| <b>DriverManager</b> | Finds and connects the correct database driver |
|----------------------|------------------------------------------------|

|                   |                                             |
|-------------------|---------------------------------------------|
| <b>Connection</b> | Represents the link between Java and the DB |
|-------------------|---------------------------------------------|

|                  |                                  |
|------------------|----------------------------------|
| <b>try-catch</b> | Handles connection errors safely |
|------------------|----------------------------------|

|            |                                      |
|------------|--------------------------------------|
| <b>URL</b> | Describes database location and name |
|------------|--------------------------------------|

---

Would you like me to show what happens *internally* when `DriverManager.getConnection()` is called (step-by-step under the hood)?

Excellent question, Nandhini  — you've now seen **three JDBC connection styles**, and it's great that you want to understand their differences clearly. Let's break them down step by step 

---

### 1 Old (Manual) Way — With finally Block

```
Connection conn = null;  
  
try {  
  
    Class.forName("com.mysql.cj.jdbc.Driver"); // Optional now  
  
    conn = DriverManager.getConnection(url, user, pass);  
  
    System.out.println("Connected!");  
  
} catch (Exception e) {  
  
    System.out.println("Error: " + e.getMessage());  
  
} finally {  
  
    if (conn != null) {
```

```
        try { conn.close(); }

        catch (SQLException e) { System.out.println(e.getMessage()); }

    }

}
```

### **Explanation:**

- **Driver is loaded manually** using Class.forName()
- **Connection closed manually** inside a finally block
- Common in older Java versions (Java 6 or below)
- Ensures connection is always closed even if an error occurs

### **Pros:**

- ✓ Backward compatible with older JDBC versions
- ✓ Explicit control over connection closing

### **Cons:**

- ✗ More code, harder to read
  - ✗ Easy to forget to close connections
  - ✗ More error-prone
- 
- 

We use this only because it is encounter the error

### **2 Modern Try-with-Resources (Ideal Way)**

```
try (Connection conn = DriverManager.getConnection(url, user, pass)) {

    if (conn != null) {

        System.out.println("Connected to the database!");

    }

} catch (Exception e) {

    System.out.println("Connection Failed...");

}
```

### **Explanation:**

- Uses **try-with-resources** (introduced in Java 7)
- Automatically closes the connection when the try block ends
- No need for a finally block

### **Pros:**

- ✓ Clean and short code
- ✓ Connection automatically closed (no memory leak)
- ✓ Modern and recommended for production

### **Cons:**

- ✗ Slightly confusing for beginners at first glance
  - ✗ Works only for resources that implement AutoCloseable (which Connection does)
- 

### **3 Simple Try-Catch (Without Closing Connection)**

```
try {  
    Connection conn = DriverManager.getConnection(url, user, pass);  
    if (conn != null) {  
        System.out.println("Connected to the database!");  
    }  
} catch (Exception e) {  
    System.out.println("Connection Failed...");  
}
```

### **Explanation:**

- Simplest structure — connects and prints messages
- **Does not close the connection**

### **Pros:**

- ✓ Very simple for quick testing or demos
- ✓ Good for beginners to understand basic connection logic

### **Cons:**

**X** Does **not close connection** — can cause memory leaks if used repeatedly

**X** Not suitable for real-world applications

---

### Summary Table

| Version      | Style              | Closes Connection                          | Recommended                                                                                                               | Used In              |
|--------------|--------------------|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|----------------------|
| 1 Old Way    | try-catch-finally  | <input checked="" type="checkbox"/> Manual | For backward compatibility                                                                                                | Legacy code (Java 6) |
| 2 Modern Way | try-with-resources | <input checked="" type="checkbox"/> Auto   | <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> Best practice | Modern Java apps     |
| 3 Simple Try | Basic try-catch    | <input checked="" type="checkbox"/> No     | <input checked="" type="checkbox"/> Not recommended                                                                       | Learning/demo only   |

---

### In short

👉 The **first code** is traditional (manual close).

👉 The **second code** is **modern and ideal** (auto-close).

👉 The **third code** is **basic demo** (no close).

---

Would you like me to show you a **fourth version** — the *fully professional JDBC template* version that includes executing queries (Statement + ResultSet) inside try-with-resources?

---

Sure, Nandhini 🌸 — here's the  **Summary Table** explained in **clear point form** 👇

---

#### 1 Old Way — try-catch-finally

- ⚡ Uses **manual connection closing** inside finally block
- ⚡ Works in **older Java versions (like Java 6)**
- ⚡ Ensures resources are closed even if an error occurs
- ⚡  Connection closed manually using conn.close()

- Used in **legacy applications**
  - More code and higher chance of mistakes
  - Not preferred for modern Java
- 

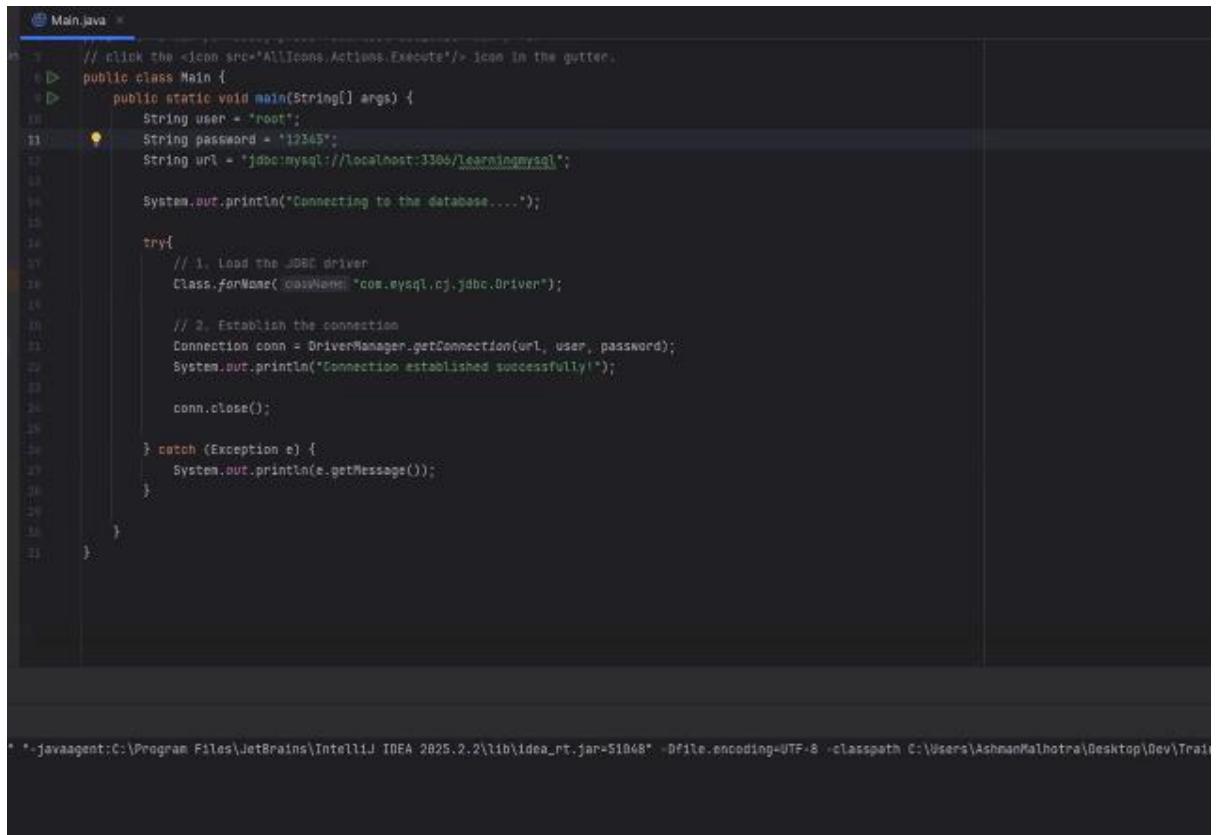
## **2 Modern Way — try-with-resources**

- Uses **Java 7+ feature**: try-with-resources
  - Connection is **closed automatically** after try block ends
  - No need for finally or manual closing
  - Clean, short, and safe — **Best Practice**
  - Used in **modern Java applications**
  - Prevents memory leaks automatically
- 

## **3 Simple Try — Basic try-catch**

- Creates connection but **never closes it**
  - Can lead to **memory leaks** or **open connections**
  - Used only for **learning or demo purposes**
  - Easy to understand for beginners
  - Not suitable for real projects
- 

Would you like me to make a small **diagram or visual flow** showing how connection closes differently in these three cases? It can help you understand the life cycle better.



The screenshot shows the IntelliJ IDEA interface with a Java file named Main.java open. The code is a simple program that connects to a MySQL database. It includes imports for Connection and DriverManager, defines a Main class with a main method, and uses System.out.println to output connection status. A try-catch block handles JDBC driver loading and connection establishment. The code is annotated with comments explaining the steps: // 1. Load the JDBC driver, // 2. Establish the connection, and // 3. Close the connection. The code is well-formatted with proper indentation and line breaks.

```
// click the <Icon src="AllIcons.Actions.Execute"/> icon in the gutter.
public class Main {
    public static void main(String[] args) {
        String user = "root";
        String password = "12345";
        String url = "jdbc:mysql://localhost:3306/learningmysql";

        System.out.println("Connecting to the database....");

        try{
            // 1. Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // 2. Establish the connection
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("Connection established successfully!");

            conn.close();

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

*-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2025.2.2\lib\idea_rt.jar=S1048* -Dfile.encoding=UTF-8 -classpath C:\Users\AshmanMalhotra\Desktop\Dev\Train
```

```
import java.sql.Connection;
import java.sql.DriverManager;

public class Main {
    public static void main(String[] args) {
        String user = "root";
        String password = "12345";
        String url = "jdbc:mysql://localhost:3306/learningmysql";

        System.out.println("Connecting to the database...");

        try {
            // 1. Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // 2. Establish the connection
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("Connection established successfully!");

            // 3. Close the connection
            conn.close();
        } catch (Exception e) {
```

```

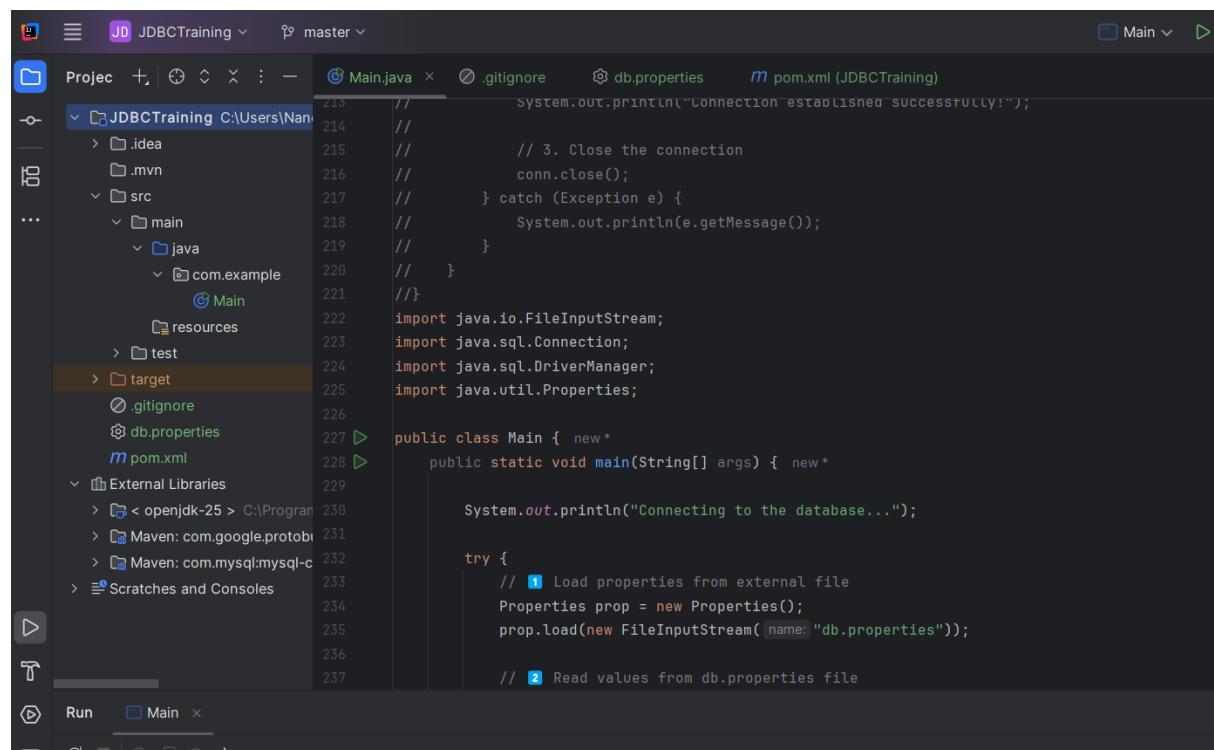
        System.out.println(e.getMessage());
    }
}
}

```

```

dburl=jdbc:mysql://localhost:3306/learningmysql
user=root
password=12345

```



```

import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {
        System.out.println("Connecting to the database...");
    }
}

```

```

try {
    // 1 Load properties from external file
    Properties prop = new Properties();
    prop.load(new FileInputStream("db.properties"));

    // 2 Read values from db.properties file
    String url = prop.getProperty("dburl");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    // 3 Load the JDBC driver manually
    Class.forName("com.mysql.cj.jdbc.Driver");

    // 4 Establish the connection
    Connection conn = DriverManager.getConnection(url, user, password);
    System.out.println("Connection established successfully!");

    // 5 Close the connection
    conn.close();

} catch (Exception e) {
    System.out.println(e.getMessage());
}
}
}

```

Perfect, Nandhini 🌸 — this is one of the most **professional and flexible ways** to connect to a database in Java using JDBC. Let's go through your code **step-by-step in simple beginner terms** 👇

---

## Program Purpose

This program connects your Java application to a **MySQL database** using credentials (URL, username, password) that are **stored separately** in a file called db.properties.

This method is preferred because you can easily **change database details** without editing your Java code.

---

## Step-by-Step Explanation

## Step 1: Import Required Packages

```
import java.io.InputStream;  
  
import java.sql.Connection;  
  
import java.sql.DriverManager;  
  
import java.util.Properties;
```

- `InputStream` → Helps read files from your system.
  - `Connection` → Represents an active link between your Java program and the database.
  - `DriverManager` → Manages the drivers and establishes a connection.
  - `Properties` → Reads key-value pairs from a `.properties` file (like `dburl`, `user`, `password`).
- 

## Step 2: Print a Start Message

```
System.out.println("Connecting to the database...");
```

Just prints a message to show that the connection process is starting.

---

## Step 3: Load Properties File

```
Properties prop = new Properties();  
  
prop.load(new FileInputStream("db.properties"));
```

- Creates a `Properties` object named `prop`.
  - Reads the file **db.properties** which contains your connection info.
  - The file might look like this 
  - `dburl=jdbc:mysql://localhost:3306/learningmysql`
  - `user=root`
  - `password=12345`
  - Each line is stored as a **key-value pair** inside the `prop` object.
- 

## Step 4: Read the Values

```
String url = prop.getProperty("dburl");
String user = prop.getProperty("user");
String password = prop.getProperty("password");
```

- Extracts the values for each key from the properties file.
  - Stores them in Java variables url, user, and password.
  - These values will be used to connect to the database.
- 

## 5 Step 5: Load JDBC Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- This **loads the MySQL JDBC driver** class into memory.
  - It tells Java which database type you're connecting to (MySQL here).
  - Newer JDBC versions can do this automatically, but adding it is still good practice.
- 

## 6 Step 6: Establish Connection

```
Connection conn = DriverManager.getConnection(url, user, password);
```

```
System.out.println("Connection established successfully!");
```

- DriverManager.getConnection() actually **connects** your Java program to the MySQL database.
  - If successful, it returns a Connection object (conn) which represents the open connection.
  - You can now use conn to run SQL queries (like SELECT, INSERT, UPDATE, DELETE).
- 

## 7 Step 7: Close Connection

```
conn.close();
```

- Once the work is done, you **close the connection** to free resources.
  - Always important to avoid memory leaks or open connections.
-

## Step 8: Handle Errors

```
catch (Exception e) {  
    System.out.println(e.getMessage());  
}
```

- Catches any errors (like wrong password, database not running, or missing driver).
  - Prints the error message to the console.
- 

### In Short

 You are:

1. Reading database details from an external file.
  2. Loading the MySQL JDBC driver.
  3. Establishing a connection to the MySQL database.
  4. Printing confirmation and closing the connection safely.
- 

### Why This Method Is Used

- You can easily change credentials without touching your code.
  - Secure — passwords are not hardcoded in Java.
  - Clean, organized, and professional approach.
  - Works well for larger applications or multiple databases.
- 

Would you like me to show the **next version** of this code using **try-with-resources**, so that even if an error occurs, the connection closes automatically?

```
import java.io.FileInputStream;  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.util.Properties;  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Connecting to the database...");
```

```

// Step 1: Create a Properties object
Properties prop = new Properties();

// Step 2: Load db.properties file safely using try-with-resources
try (FileInputStream fis = new FileInputStream("db.properties")) {
    prop.load(fis);
} catch (Exception e) {
    System.out.println("Error loading properties file: " + e.getMessage());
    return; // Exit if properties not loaded
}

// Step 3: Get database credentials from file
String url = prop.getProperty("dburl");
String user = prop.getProperty("user");
String password = prop.getProperty("password");

// Step 4: Try to connect to the database
try (Connection conn = DriverManager.getConnection(url, user, password)) {
    if (conn != null) {
        System.out.println(" Connected to the database successfully!");
    } else {
        System.out.println(" Connection object is null!");
    }
} catch (Exception e) {
    System.out.println("Connection Failed: " + e.getMessage());
}
}
}

```

Excellent question, Nandini 🙌 — both versions connect to the same database successfully, but they differ mainly in **code style**, **safety**, and **best practices**. Let's break it down clearly 👇

---

### Difference Between the Two Codes

| Feature                | ◆ Old Version                                                                                                           | ◆ Improved Version (Final)                                                                       |
|------------------------|-------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| 1 File Handling        | Uses normal FileInputStream inside try block.<br>Must be closed manually (not done).                                    | Uses <b>try-with-resources</b> , which automatically closes the file stream after use.           |
| 2 Connection Closing   | Closes connection manually using conn.close().<br><b>!</b> If exception occurs before that line, connection stays open. | Uses <b>try-with-resources</b> for Connection, so it auto-closes safely even if an error occurs. |
| 3 JDBC Driver Loading  | Explicitly loads driver using Class.forName("com.mysql.cj.jdbc.Driver").                                                | Not needed anymore (JDBC 4+ auto-loads driver). Cleaner and modern.                              |
| 4 Error Handling       | Single try-catch for all — if any step fails, it may not be clear which one.                                            | Separate try-blocks for loading file and connecting DB → more clear error reporting.             |
| 5 Readability & Safety | Works fine but less safe and slightly outdated.                                                                         | Clean, safe, and recommended in modern Java (Java 7+).                                           |

## ✳️ In Simple Words

- 👉 **Old Code** → Works fine but uses *manual resource management*.
- 👉 **New Code** → Uses *automatic resource management* and *modern JDBC style* (simpler + safer).

## ✅ Which One to Use?

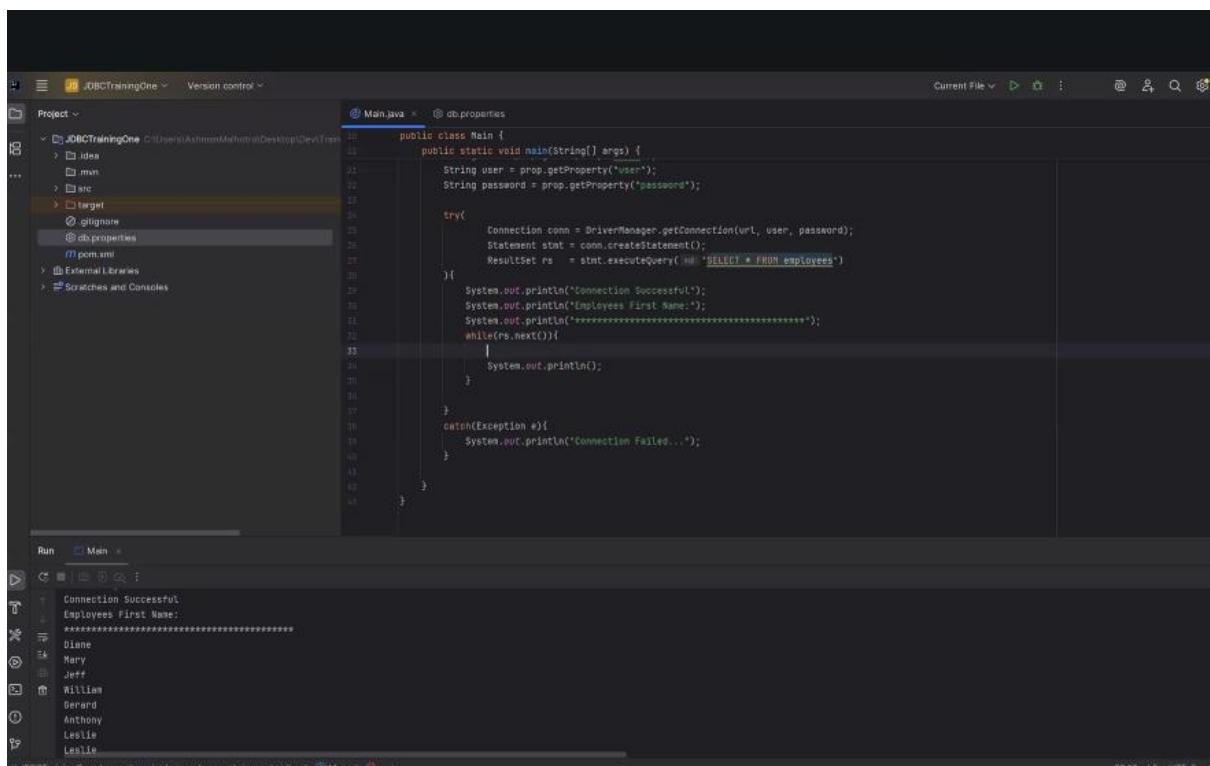
Always prefer the **modern version (with try-with-resources)** in real-world projects.  
It ensures no resource leaks and follows the latest Java best practices.

---

Would you like me to show a **side-by-side diagram** comparing how resource closing works internally (old vs new)?

The **first code** loads the JDBC driver manually using `Class.forName()` and doesn't use `try-with-resources`, so connections and streams must be closed manually.

The **second code** uses **try-with-resources** for safer resource handling and **automatically closes** the `FileInputStream` and `Connection`, removing the need to load the driver explicitly.



The screenshot shows an IDE interface with a project named "JDBCTrainingOne". The "Main.java" file is open, displaying Java code that connects to a database and prints employee names. The "Run" tab shows the output of the program, which includes the connection message and a list of employee first names. The employee names listed are Diane, Mary, Jeff, William, Gerard, Anthony, Leslie, and Leslie.

```
public class Main {
    public static void main(String[] args) {
        String user = prop.getProperty("user");
        String password = prop.getProperty("password");

        try{
            Connection conn = DriverManager.getConnection(url, user, password);
            Statement stat = conn.createStatement();
            ResultSet rs   = stat.executeQuery("SELECT * FROM employees");
        }{
            System.out.println("Connection Successful!");
            System.out.println("Employee First Name:");
            System.out.println("*****");
            while(rs.next()){
                System.out.println();
            }
        }
        catch(Exception e){
            System.out.println("Connection Failed...");}
    }
}
```

Run Main  
Connection Successful:  
Employee First Name:  
\*\*\*\*\*  
Diane  
Mary  
Jeff  
William  
Gerard  
Anthony  
Leslie  
Leslie

```
public class Main {
    public static void main(String[] args) {
        String user = prop.getProperty("user");
        String password = prop.getProperty("password");

        try{
            Connection conn = DriverManager.getConnection(url, user, password);
            Statement stmt = conn.createStatement();
            ResultSet rs   = stmt.executeQuery( sql "SELECT * FROM employees");
        }{
            System.out.println("Connection Successful");
            System.out.println("Employees First Name:");
            System.out.println("*****");
            while(rs.next()){
                String fname = rs.getString( columnLabel: "firstName");
                String empNo = rs.getString( columnLabel: "employeeNumber");
                String empNo = rs.getString( columnLabel: "jobTitle");

                System.out.println();
            }
        }
        catch(Exception e){
            System.out.println("Connection Failed...");
        }
    }
}
```

```
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {

        System.out.println("Connecting to the database...");

        // Step 1: Create a Properties object
        Properties prop = new Properties();

        // Step 2: Load db.properties file safely using try-with-resources
        try (FileInputStream fis = new FileInputStream("db.properties")){
            prop.load(fis);
        } catch (Exception e){
            System.out.println("Error loading properties file: " + e.getMessage());
            return; // Exit if properties not loaded
        }
    }
}
```

```

// Step 3: Get database credentials from file
String url = prop.getProperty("dburl");
String user = prop.getProperty("user");
String password = prop.getProperty("password");

// Step 4: Connect to database and fetch employee details
try (Connection conn = DriverManager.getConnection(url, user, password)) {
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM employees") {

        System.out.println("Connection Successful!");
        System.out.println("Employees Data:");
        System.out.println("*****");

        // Step 5: Display data
        while (rs.next()) {
            String fname = rs.getString("firstName");
            String empNo = rs.getString("employeeNumber");
            String jtitle = rs.getString("jobTitle");

            System.out.println("Employee No: " + empNo);
            System.out.println("First Name : " + fname);
            System.out.println(empNo + ", " + fname + ", " + jtitle);

        }
    } catch (Exception e) {
        System.out.println("Connection Failed: " + e.getMessage());
    }
}

```

Perfect 👍 Nandhini! Let's go through your Java JDBC code **line by line in very simple beginner-friendly words** 🤝

---

### Import Statements

```

import java.io.FileInputStream;
import java.sql.Connection;

```

```
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.Statement;  
import java.util.Properties;
```

- These lines **import classes** from Java's library.
  - FileInputStream → reads the db.properties file (which stores database info).
  - Properties → stores key-value pairs (like username and password).
  - Connection, Statement, ResultSet, DriverManager → used for **JDBC (Java Database Connectivity)** to connect and run SQL queries.
- 

## Main Class and Method

```
public class Main {  
  
    public static void main(String[] args) {  
  
        • This is your main class and starting point of the program.  
  
        • main() method runs automatically when you execute the program.  
  
    }  
}
```

---

## Print Starting Message

```
System.out.println("Connecting to the database...");
```

- Just prints a message so you know what's happening.
- 

## Create a Properties Object

```
Properties prop = new Properties();
```

- Creates a Properties object that will **hold your database settings** from the file.
- 

## Load db.properties File

```
try (FileInputStream fis = new FileInputStream("db.properties")) {  
  
    prop.load(fis);  
  
} catch (Exception e) {
```

```
System.out.println("Error loading properties file: " + e.getMessage());  
return;  
}
```

- Opens the file named **db.properties**.
  - Loads its contents (like dburl, user, password) into the prop object.
  - If file is missing or has an error, it prints the message and **stops the program**.
- 

## **Read Database Credentials**

```
String url = prop.getProperty("dburl");  
  
String user = prop.getProperty("user");  
  
String password = prop.getProperty("password");
```

- Reads values from the file, like:
  - dburl=jdbc:mysql://localhost:3306/your\_database
  - user=root
  - password=1234
- 

## **Connect and Execute Query**

```
try (Connection conn = DriverManager.getConnection(url, user, password);  
     Statement stmt = conn.createStatement();  
     ResultSet rs = stmt.executeQuery("SELECT * FROM employees")) {  
  
    • DriverManager.getConnection() → connects your Java program to the database  
      using the URL, username, and password.  
  
    • Statement → used to write SQL commands.  
  
    • executeQuery() → runs the SQL command and stores results in a ResultSet  
      object.  
    (Here it runs SELECT * FROM employees — fetches all employee records.)
```

---

## **Connection Successful Message**

```
System.out.println("Connection Successful!");
```

```
System.out.println("Employees Data:");
System.out.println("*****");
```

- Prints confirmation that connection worked and a heading before showing data.
- 

## Loop Through Results

```
while (rs.next()) {
    String fname = rs.getString("firstName");
    String empNo = rs.getString("employeeNumber");
    String jtitle = rs.getString("jobTitle");
```

- `rs.next()` → moves to the next row in the result table.
  - `getString("columnName")` → fetches the value of a specific column by its name (like `firstName`).
  - The data is stored in variables.
- 

## Display the Data

```
System.out.println("Employee No: " + empNo);
System.out.println("First Name : " + fname);
System.out.println(empNo + ", " + fname + ", " + jtitle);
    • Prints each employee's number, name, and job title neatly.
```

Example output:

```
Employee No: 1001
First Name : Nandhini
1001, Nandhini, Developer
```

---

## Handle Errors

```
} catch (Exception e) {
    System.out.println("Connection Failed: " + e.getMessage());
}
```

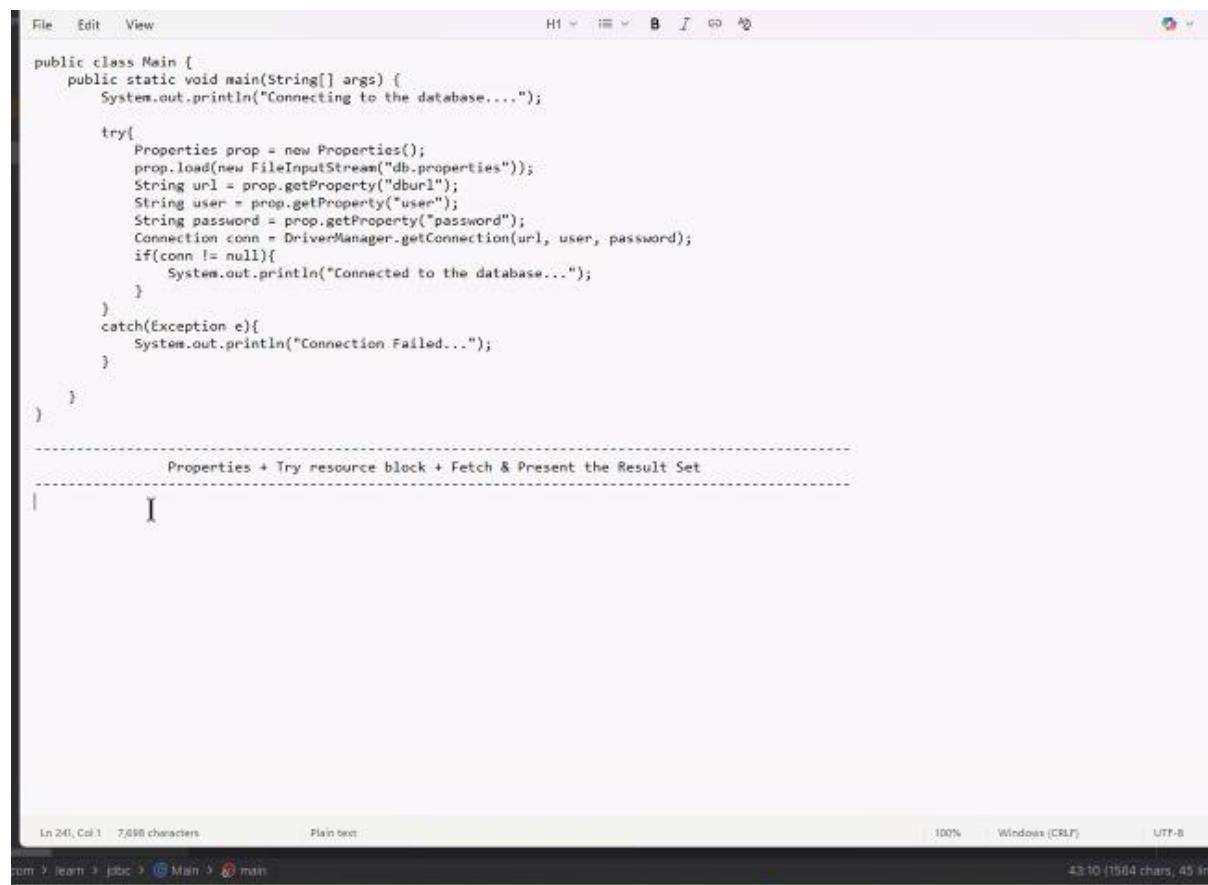
- If any problem occurs (like wrong credentials or server down), it prints the error message.
- 

### Summary:

| Step | What it Does                                        | Example                 |
|------|-----------------------------------------------------|-------------------------|
| 1    | Reads database info from file dburl, user, password |                         |
| 2    | Connects to MySQL                                   | Using DriverManager     |
| 3    | Runs SQL query                                      | SELECT * FROM employees |
| 4    | Reads each row                                      | rs.next()               |
| 5    | Prints details                                      | empNo, fname, jtitle    |

---

Would you like me to also show what your db.properties file should look like for this code?



```

File Edit View H1 I= B I= E0 A0
public class Main {
    public static void main(String[] args) {
        System.out.println("Connecting to the database....");

        try{
            Properties prop = new Properties();
            prop.load(new FileInputStream("db.properties"));
            String url = prop.getProperty("dburl");
            String user = prop.getProperty("user");
            String password = prop.getProperty("password");
            Connection conn = DriverManager.getConnection(url, user, password);
            if(conn != null){
                System.out.println("Connected to the database...");
            }
        } catch(Exception e){
            System.out.println("Connection Failed...");
        }
    }
}

----- Properties + Try resource block + Fetch & Present the Result Set -----
| |

```

Ln 241, Col 1 7,698 characters Plain text 100% Windows (CRLF) UTT-B

com > learn > jdbc > Main.java main 43:10 (1564 chars/ 45 line)

```
>Main.java  db.properties

1 package com.learn.jdbc;
2
3 import com.mysql.cj.jdbc.MysqlDataSource;
4
5 import java.io.FileInputStream;
6 import java.util.Properties;
7
8 public class Main {
9     public static void main(String[] args) {
10         Properties prop = new Properties();
11         try(FileInputStream fis = new FileInputStream( name: "db.properties")){
12             prop.load(fis);
13         } catch (Exception e) {
14             System.out.println(e.getMessage());
15         }
16
17         String url = prop.getProperty("dburl");
18         String user = prop.getProperty("user");
19         String password = prop.getProperty("password");
20
21         MysqlDataSource dataSource = new MysqlDataSource();
22         dataSource.setURL(url);
23         dataSource.setUser(user);
24         dataSource.setPassword(password);
25
26         System.out.println("connecting to the database....");
27     }
28 }
```

```
>Main.java  db.properties
1 package com.learn.jdbc;
2
3 import com.mysql.cj.jdbc.DataSource;
4
5 import java.io.FileInputStream;
6 import java.sql.Connection;
7 import java.util.Properties;
8
9 public class Main {
10     public static void main(String[] args) {
11         Properties prop = new Properties();
12         try(FileInputStream fis = new FileInputStream("db.properties")){
13             prop.load(fis);
14         } catch (Exception e) {
15             System.out.println(e.getMessage());
16         }
17
18         String url = prop.getProperty("url");
19         String user = prop.getProperty("user");
20         String password = prop.getProperty("password");
21
22         MysqlDataSource dataSource = new MysqlDataSource();
23         dataSource.setURL(url);
24         dataSource.setUser(user);
25         dataSource.setPassword(password);
26
27         try(Connection conn = dataSource.getConnection()){
28             System.out.println("Connected to database successfully");
29         } catch (Exception e){
30             System.out.println(e.getMessage());
31         }
32
33     }
34 }
```

DataSource is an **improved way** to connect your Java program to a database.  
It's like a **connection manager** that gives you database connections **safely and efficiently** — better than the old DriverManager.

```
import com.mysql.cj.jdbc.DataSource;
import java.io.FileInputStream;
import java.sql.Connection;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {

        Properties prop = new Properties(); // ✅ Move this outside the try block

        // Step 1: Load db.properties file
        try (FileInputStream fis = new FileInputStream("db.properties")){
            prop.load(fis);
        } catch (Exception e){
            System.out.println("Error loading properties file: " + e.getMessage());
            return; // Exit if file not loaded
        }
    }
}
```

```

        }

// Step 2: Read values from the properties file
String url = prop.getProperty("dburl");
String user = prop.getProperty("user");
String password = prop.getProperty("password");

// Step 3: Configure the MySQL DataSource
MysqlDataSource dataSource = new MysqlDataSource();
dataSource.setURL(url);
dataSource.setUser(user);
dataSource.setPassword(password);

// Step 4: Connect to the database
try (Connection con = dataSource.getConnection()) {
    System.out.println("Connected to database successfully!");
} catch (Exception e) {
    System.out.println("Connection Failed: " + e.getMessage());
}
}
}
}

```

in this code we need to extract data from database

```

import com.mysql.cj.jdbc.MysqlDataSource;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {

        // Step 1: Load properties file
        Properties prop = new Properties();

        try (FileInputStream fis = new FileInputStream("db.properties")) {
            prop.load(fis);
        } catch (Exception e) {
            System.out.println("Error loading properties file: " + e.getMessage());
            return;
        }
    }
}

```

```

}

// Step 2: Read database details
String url = prop.getProperty("dburl");
String user = prop.getProperty("user");
String password = prop.getProperty("password");

// Step 3: Create a MySQLDataSource object
MysqlDataSource dataSource = new MysqlDataSource();
dataSource.setURL(url);
dataSource.setUser(user);
dataSource.setPassword(password);

System.out.println("Connecting to the database...");

// Step 4: Establish connection and query data
try (Connection conn = dataSource.getConnection();
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT firstName, employeeNumber, jobTitle
FROM employees")) {

    System.out.println("Connection Successful!");
    System.out.println("*****");

    while (rs.next()) {
        String fname = rs.getString("firstName");
        String empNo = rs.getString("employeeNumber");
        String jtitle = rs.getString("jobTitle");

        System.out.println(empNo + ", " + fname + ", " + jtitle);
    }
}

} catch (Exception e) {
    System.out.println("Connection Failed: " + e.getMessage());
}
}
}

```

Excellent question, Nandhini 🌟 — both versions of your code do **the same core task** (connect to MySQL using db.properties), but there are a few **key differences in functionality, structure, and purpose**. Let's go line by line to understand clearly 👇

---

### ✳️ 1. Functionality

| Feature        | First Code                                                | Second Code                                                                                             |
|----------------|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Purpose        | Only connects to the database and prints success/failure. | Connects to the database <b>and retrieves employee data</b> (from employees table).                     |
| SQL Query      | ✗ No query executed.                                      | <input checked="" type="checkbox"/> Executes SELECT firstName, employeeNumber, jobTitle FROM employees. |
| Result Display | Only “Connected successfully!”                            | Displays each employee’s number, name, and job title.                                                   |

---

### ✳️ 2. Structure

| Aspect              | First Code                                          | Second Code                                                            |
|---------------------|-----------------------------------------------------|------------------------------------------------------------------------|
| Imports             | Minimal – no ResultSet or Statement.                | Includes ResultSet and Statement for querying.                         |
| Code Length         | Shorter and simpler.                                | Longer because it processes query results.                             |
| Resource Management | Uses try-with-resources for file and DB connection. | Same, but also includes Statement and ResultSet in try-with-resources. |

---

### ⚙️ 3. Usage Purpose

| Type                               | Description                                                                                                                              |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| First Code (Basic Connection Test) | Used mainly to <b>test database connectivity</b> using a DataSource object. It ensures that credentials in db.properties work correctly. |

| Type                                              | Description                                                                                            |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <b>Second Code<br/>(Functional Query<br/>App)</b> | Used for <b>retrieving and displaying data</b> from a MySQL table — a real-world application scenario. |

---

## 4. Output Example

### First Code Output:

Connected to database successfully!

### Second Code Output:

Connection Successful!

\*\*\*\*\*

1002, John, Sales Manager

1056, Maria, Accountant

1078, Raj, Developer

...

---

## In summary

| Feature                      | First Code           | Second Code              |
|------------------------------|----------------------|--------------------------|
| Connect to DB                | ✓                    | ✓                        |
| Execute SQL query            | ✗                    | ✓                        |
| Display table data           | ✗                    | ✓                        |
| Use of Statement & ResultSet | ✗                    | ✓                        |
| For learning/testing         | ✓ Best for beginners | ✓ Best for practical use |

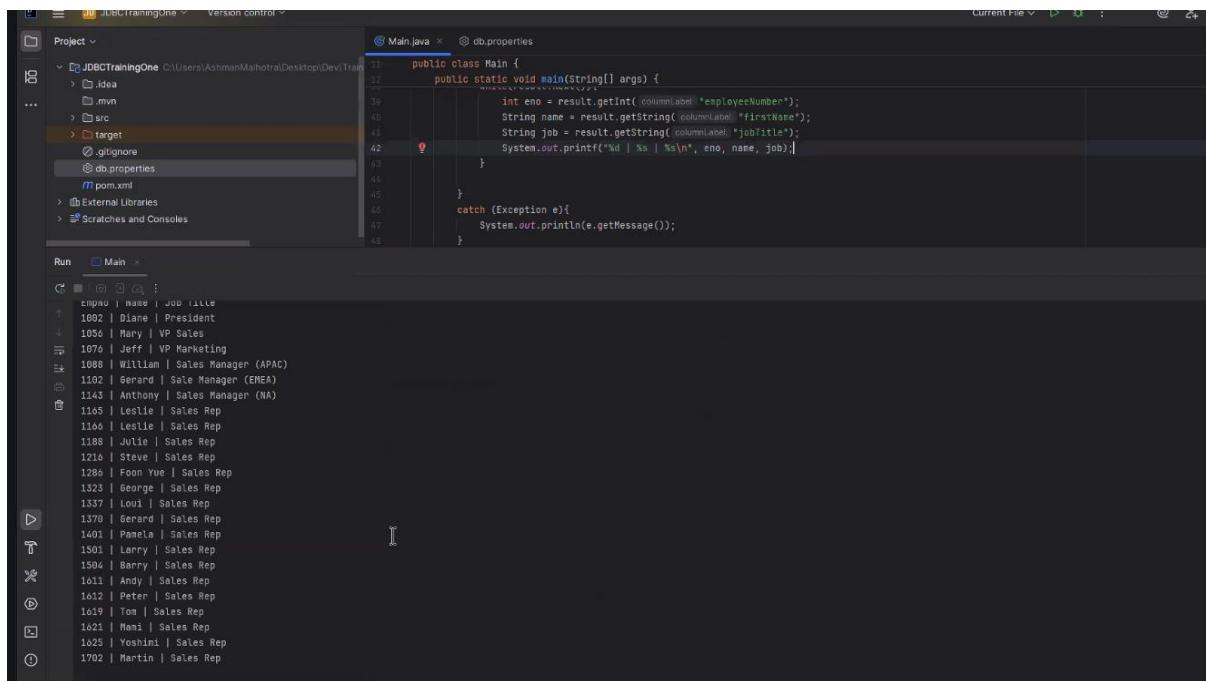
---

Would you like me to show how to **add user input** (e.g., search employee by job title or name) to the second program? It's a good next step after this.

```
 1 Main.java - db.properties
 2
 3 import com.mysql.cj.jdbc.DataSource;
 4
 5 import java.io.FileInputStream;
 6 import java.sql.Connection;
 7 import java.sql.ResultSet;
 8 import java.sql.Statement;
 9 import java.util.Properties;
10
11 public class Main {
12     public static void main(String[] args) {
13         Properties prop = new Properties();
14         try(FileInputStream fis = new FileInputStream( name "db.properties")){
15             prop.load(fis);
16         } catch (Exception e) {
17             System.out.println(e.getMessage());
18         }
19
20         String url = prop.getProperty("dburl");
21         String user = prop.getProperty("user");
22         String password = prop.getProperty("password");
23         String query   = "SELECT * FROM employees";
24
25         MysqlDataSource dataSource = new MysqlDataSource();
26         dataSource.setURL(url);
27         dataSource.setUser(user);
28         dataSource.setPassword(password);
29
30         try{
31             Connection conn = dataSource.getConnection();
32             Statement stmt = conn.createStatement();
33             ResultSet result = stmt.executeQuery(query);
34
35         }{
36             System.out.println("Connected to database successfully");
37             System.out.println("Employee Number | Name | Job Title");
38             while(result.next()){
39
40             }
41         }
42     } catch (Exception e){
43         System.out.println(e.getMessage());
44     }
45 }
46
47
48
49
50
51 }
52 }
```

```
 1 Main.java x db.properties
 2
 3 public class Main {
 4     public static void main(String[] args) {
 5         prop.load(fis);
 6     } catch (Exception e) {
 7         System.out.println(e.getMessage());
 8     }
 9
10     String url = prop.getProperty("dburl");
11     String user = prop.getProperty("user");
12     String password = prop.getProperty("password");
13     String query   = "SELECT * FROM employees";
14
15     MysqlDataSource dataSource = new MysqlDataSource();
16     dataSource.setURL(url);
17     dataSource.setUser(user);
18     dataSource.setPassword(password);
19
20     try{
21         Connection conn = dataSource.getConnection();
22         Statement stmt = conn.createStatement();
23         ResultSet result = stmt.executeQuery(query);
24
25     }{
26         System.out.println("Connected to database successfully");
27         System.out.println("EmpNo | Name | Job Title");
28         while(result.next()){
29             int eno = result.getInt( columnLabel: "employeeNumber");
30             String name = result.getString( columnLabel: "firstName");
31             String job = result.getString( columnLabel: "jobTitle");
32             System.out.printf("%d | %s | %s\n", eno, name, job);
33         }
34     }
35 }
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52 }
```

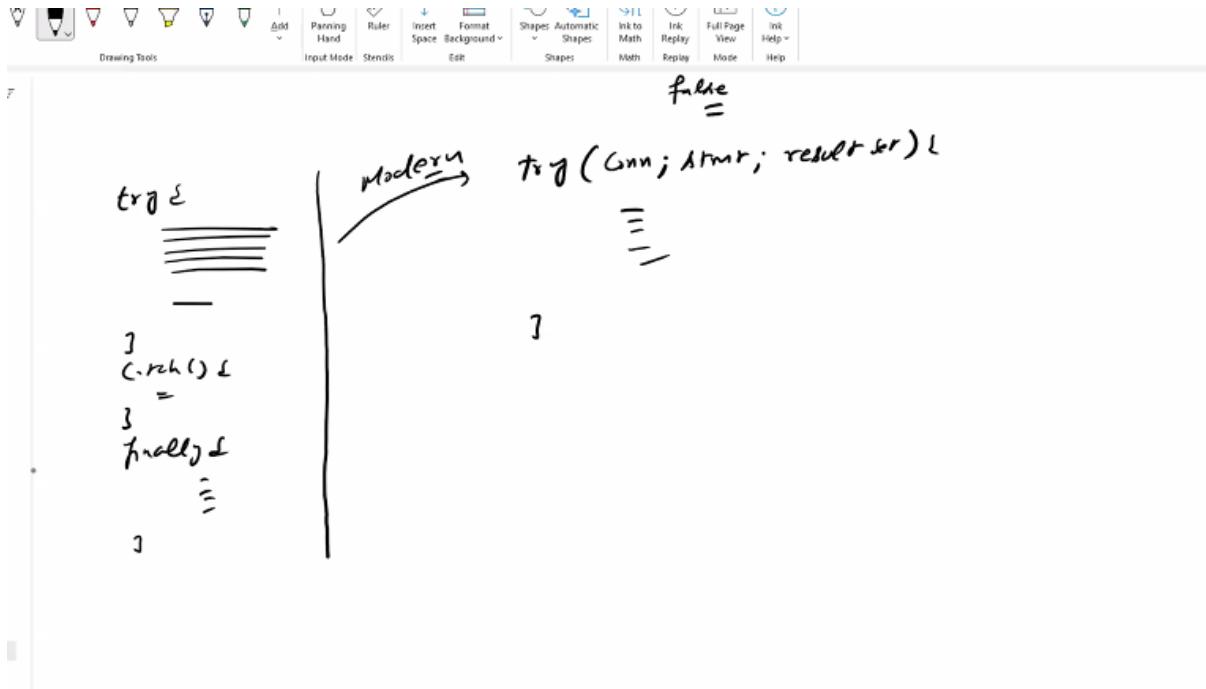
```
>Main.java x db.properties
1 import java.sql.ResultSet;
2 import java.sql.Statement;
3 import java.util.Properties;
4
5 public class Main {
6     public static void main(String[] args) {
7         Properties prop = new Properties();
8         try(FileInputStream fis = new FileInputStream("name:db.properties")){
9             prop.load(fis);
10        } catch (Exception e) {
11            System.out.println(e.getMessage());
12        }
13
14        String url = prop.getProperty("dburl");
15        String user = prop.getProperty("user");
16        String password = prop.getProperty("password");
17        String query    = "SELECT * FROM employees";
18
19        MySQLDataSource dataSource = new MySQLDataSource();
20        dataSource.setURL(url);
21        dataSource.setUser(user);
22        dataSource.setPassword(password);
23
24        try{
25            Connection conn   = dataSource.getConnection();
26            Statement stmt   = conn.createStatement();
27            ResultSet result = stmt.executeQuery(query);
28
29        }{
30            System.out.println("Connected to database successfully");
31            System.out.println("EmpNo | Name | Job Title");
32            while(result.next()){
33                int eno = result.getInt(columnLabel: "employeeNumber");
34                String name = result.getString(columnLabel: "firstName");
35                String job = result.getString(columnLabel: "jobTitle");
36                System.out.printf("%d | %s | %s\n", eno, name, job);
37            }
38        }
39        catch (Exception e){
40            System.out.println(e.getMessage());
41        }
42    }
43}
```



The screenshot shows the Java IDE interface with the following details:

- Project View:** Shows the project structure under "JDBCTrainingOne". The "target" folder is selected.
- Main.java:** The code is identical to the one shown in the previous screenshot.
- Output Window:** Displays the results of the program execution:

```
EmpNo | Name | Job Title
1002 | Diane | President
1056 | Mary | VP Sales
1076 | Jeff | VP Marketing
1088 | William | Sales Manager (APAC)
1102 | Gerard | Sales Manager (EMEA)
1143 | Anthony | Sales Manager (NA)
1165 | Leslie | Sales Rep
1166 | Leslie | Sales Rep
1188 | Julie | Sales Rep
1216 | Steve | Sales Rep
1286 | Foon Yee | Sales Rep
1323 | George | Sales Rep
1337 | Loui | Sales Rep
1370 | Gerard | Sales Rep
1401 | Pamela | Sales Rep
1501 | Larry | Sales Rep
1504 | Barry | Sales Rep
1611 | Andy | Sales Rep
1612 | Peter | Sales Rep
1619 | Tom | Sales Rep
1621 | Mani | Sales Rep
1625 | Yoshimi | Sales Rep
1702 | Martin | Sales Rep
```



Excellent, Nandhini 🌸 — this is a **very common and important interview question!**

Let's clearly understand the **difference between Traditional JDBC and Modern JDBC (DataSource-based)** 👇

### Traditional JDBC vs Modern JDBC

| Feature                    | Traditional JDBC (DriverManager)                                          | Modern JDBC (DataSource)                                                               |
|----------------------------|---------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <b>Connection Handling</b> | Uses DriverManager to manually open and close connections.                | Uses DataSource interface, often managed by the server or framework.                   |
| <b>Code Example</b>        | <pre>Connection con = DriverManager.getConnection(url, user, pass);</pre> | <pre>DataSource ds = new MysqlDataSource(); Connection con = ds.getConnection();</pre> |
| <b>Configuration</b>       | Database URL, username, and password are hardcoded in the program.        | Configuration done externally (in XML, properties file, or server).                    |
| <b>Connection Pooling</b>  | Not supported — new connection for every request (slow).                  | Supports <b>connection pooling</b> — reuses existing connections (faster).             |

| Feature         | Traditional JDBC (DriverManager)                     | Modern JDBC (DataSource)                                |
|-----------------|------------------------------------------------------|---------------------------------------------------------|
| Performance     | Slower due to repeated connection creation.          | Faster and more efficient due to pooled connections.    |
| Maintainability | Hard to modify — must edit Java code for DB changes. | Easy to maintain — change settings outside code.        |
| Security        | Credentials visible in code.                         | Credentials stored securely in configuration or server. |
| Use Case        | Small, standalone applications.                      | Large, enterprise-level or web applications.            |
| Introduced In   | JDBC 1.0                                             | JDBC 2.0 and above                                      |

### In Simple Words

- **Traditional JDBC** → Manual, simple, good for learning and small projects.
- **Modern JDBC** → Smarter, efficient, uses DataSource, preferred in real-world enterprise systems.

### Analogy

Think of it like this 

- **Traditional JDBC** → You open a new door every time you enter a room.
- **Modern JDBC** → You use a key card that automatically manages entry and exit efficiently (connection pool).

Would you like me to give a **side-by-side code example** showing both *DriverManager* and *DataSource* connections for better clarity?

Perfect, Nandhini  — here's the **complete comparison of Traditional JDBC vs Modern JDBC (DataSource)** with clear explanation and code examples 

### Traditional JDBC vs Modern JDBC

| Feature                    | Traditional JDBC (DriverManager)                                | Modern JDBC (DataSource)                                                                   |
|----------------------------|-----------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>Connection Handling</b> | Uses DriverManager to manually open/close database connections. | Uses DataSource object to get connections (managed, efficient).                            |
| <b>Configuration</b>       | Hardcoded URL, username, and password in code.                  | Configured outside (via XML, properties, or server).                                       |
| <b>Connection Pooling</b>  | ✗ Not supported (creates new connection each time).             | <input checked="" type="checkbox"/> Supported (reuses connections for better performance). |
| <b>Performance</b>         | Slower, due to repeated connection creation.                    | Faster, as existing connections are reused.                                                |
| <b>Maintainability</b>     | Difficult to modify — must edit and recompile Java code.        | Easier — can change config externally without touching code.                               |
| <b>Security</b>            | Credentials are visible in the source code.                     | Credentials stored securely in external configuration.                                     |
| <b>Use Case</b>            | Small or simple applications.                                   | Enterprise or web-based applications (Spring, JEE, etc.).                                  |
| <b>JDBC Version</b>        | JDBC 1.0                                                        | JDBC 2.0 and later                                                                         |

---

## 1 Traditional JDBC Example (DriverManager)

```
import java.sql.*;

public class TraditionalJDBCEExample {

    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/studentdb";
        String user = "root";
        String password = "1234";

        try {
```

```
// Load the driver  
Class.forName("com.mysql.cj.jdbc.Driver");  
  
// Create a connection  
Connection con = DriverManager.getConnection(url, user, password);  
  
// Create a statement  
Statement stmt = con.createStatement();  
  
// Execute a query  
ResultSet rs = stmt.executeQuery("SELECT * FROM students");  
  
// Process result  
while (rs.next()) {  
    System.out.println(rs.getInt("id") + " - " + rs.getString("name"));  
}  
  
// Close connections  
rs.close();  
stmt.close();  
con.close();  
  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}
```

 **Explanation:**

- Connection created using DriverManager.getConnection().
  - Every time you call this, a **new database connection** is made — slower for large apps.
- 

## 2 Modern JDBC Example (Using DataSource)

```
import com.mysql.cj.jdbc.MysqlDataSource;  
  
import java.sql.*;  
  
  
public class ModernJDBCExample {  
    public static void main(String[] args) {  
        try {  
            // Create a DataSource object  
            MysqlDataSource ds = new MysqlDataSource();  
            ds.setURL("jdbc:mysql://localhost:3306/studentdb");  
            ds.setUser("root");  
            ds.setPassword("1234");  
  
            // Get a connection from the DataSource  
            Connection con = ds.getConnection();  
  
            // Create a statement  
            Statement stmt = con.createStatement();  
  
            // Execute a query  
            ResultSet rs = stmt.executeQuery("SELECT * FROM students");  
  
            // Process result  
            while (rs.next()) {
```

```

        System.out.println(rs.getInt("id") + " - " + rs.getString("name"));

    }

    // Close resources

    rs.close();

    stmt.close();

    con.close();

} catch (Exception e) {
    e.printStackTrace();
}

}
}

```

 **Explanation:**

- Connection obtained through a DataSource object.
  - In real-world use, this DataSource can be **configured in a server (Tomcat, GlassFish, etc.)** for connection pooling and security.
  - Faster, more maintainable, and scalable.
- 

 **In Short**

**Traditional JDBC** → Manual, simple, less efficient.

**Modern JDBC (DataSource)** → Smarter, faster, used in enterprise apps.

---

Would you like me to show how to configure a **connection pool** using DataSource (like how real companies use it in Spring or Tomcat)?  
import com.mysql.cj.jdbc.MysqlDataSource;

```

import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;

```

```
import java.sql.Statement;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {
        Properties prop = new Properties();

        // Step 2: Load db.properties file safely using try-with-resources
        try (FileInputStream fis = new FileInputStream("db.properties")) {
            prop.load(fis);
        } catch (Exception e) {
            System.out.println("Error loading properties file: " + e.getMessage());
            // Exit if properties not loaded
        }

        // Step 3: Get database credentials from file
        String url = prop.getProperty("dburl");
        String user = prop.getProperty("user");
        String password = prop.getProperty("password");
        String query = "SELECT * FROM employees";

        MysqlDataSource datasource = new MysqlDataSource();
        datasource.setURL(url);
        datasource.setUser(user);
        datasource.setPassword(password);
        // test this connection
        try( Connection conn = datasource.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(query)) {

            System.out.println("Connected to database successfully!");
            System.out.println("Employee Number | Name | job title");

            while(rs.next()) {

                String empNo= rs.getString("employeeNumber");
                String fname= rs.getString("firstname");
                String jTitle= rs.getString("jobTitle");
                System.out.printf("%s | %s | %s \n", empNo, fname, jTitle);
            }
        }
    }
}
```

```
        catch(Exception e){  
            System.out.println(e.getMessage());  
        }  
  
    }  
}
```

Perfect, Nandhini 🌸 — here's the **difference between Traditional JDBC and Modern JDBC (DataSource)** explained **point by point with code differences** 👇

---

## 💡 Difference Between Traditional JDBC and Modern JDBC

---

### ✳️ 1 Driver Loading

#### Traditional JDBC:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- You must manually load the driver class.
- Required in older JDBC versions.

#### Modern JDBC (DataSource):

```
MysqlDataSource ds = new MysqlDataSource();
```

- No need for Class.forName().
  - The driver is automatically loaded when you use the DataSource class.
- 

### ✳️ 2 Getting Connection

#### Traditional JDBC:

```
Connection con = DriverManager.getConnection(  
    "jdbc:mysql://localhost:3306/studentdb", "root", "1234");
```

- Uses **DriverManager**.
- Connection details are **hardcoded** in the program.

#### Modern JDBC:

```
ds.setURL("jdbc:mysql://localhost:3306/studentdb");
```

```
ds.setUser("root");  
ds.setPassword("1234");  
  
Connection con = ds.getConnection();
```

- Uses **DataSource** object.
  - Connection settings can be **externalized** (kept outside code).
- 

## 3 Connection Pooling

### Traditional JDBC:

-  Not supported.
- Every time you call `getConnection()`, a **new connection** is created → **slower**.

### Modern JDBC:

-  Supports **connection pooling** (especially when managed by a server).
  - Reuses existing connections → **faster** and more **scalable**.
- 

## 4 Performance

### Traditional JDBC:

- Each request → new connection → close → reopen.
- High overhead, not suitable for large systems.

### Modern JDBC:

- Reuses pooled connections → reduces overhead → improves performance.
- 

## 5 Configuration

### Traditional JDBC:

- Database URL, username, password written inside Java code.
- Example:
- `String url = "jdbc:mysql://localhost:3306/studentdb";`
- `String user = "root";`

- String password = "1234";
- Changes require code modification and recompilation.

### Modern JDBC:

- Configured externally (like in XML, properties file, or application server).
  - Easier to maintain and secure.
- 

## ✳️ 💡 Use Case

### Traditional JDBC:

- Best for small applications, testing, or learning.

### Modern JDBC:

- Used in **enterprise**, **Spring**, and **JEE** projects where performance and scalability matter.
- 

## ✓ In Short

| Aspect             | Traditional JDBC       | Modern JDBC (DataSource) |
|--------------------|------------------------|--------------------------|
| Driver loading     | Manual (Class.forName) | Automatic                |
| Connection         | DriverManager          | DataSource               |
| Connection pooling | Not available          | Available                |
| Performance        | Slower                 | Faster                   |
| Config changes     | Inside code            | External configuration   |
| Suitable for       | Small apps             | Large enterprise apps    |

---

Would you like me to show how to **add connection pooling (like in real-world servers)** using DataSource? It's a great next step for interview prep.

Great — I'll walk you through your entire program **line-by-line**, from scratch, in beginner-friendly language. I'll show **what each line does**, **why it's needed**, and **small tips** to make it safer and clearer.

For reference, here's the code you gave (I'll explain each numbered section below):

```
import com.mysql.cj.jdbc.MysqlDataSource;

import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {
        Properties prop = new Properties();

        // Step 2: Load db.properties file safely using try-with-resources
        try (FileInputStream fis = new FileInputStream("db.properties")) {
            prop.load(fis);
        } catch (Exception e) {
            System.out.println("Error loading properties file: " + e.getMessage());
        }
        // Exit if properties not loaded
    }

    // Step 3: Get database credentials from file
    String url = prop.getProperty("dburl");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");
    String query = "SELECT * FROM employees";

    MysqlDataSource datasource = new MysqlDataSource();
```

```

        datasource.setURL(url);

        datasource.setUser(user);

        datasource.setPassword(password);

//      test this connection

try(Connection conn = datasource.getConnection();

    Statement stmt = conn.createStatement();

    ResultSet rs = stmt.executeQuery(query)) {

    System.out.println("Connected to database successfully!");

    System.out.println("Employee Number | Name | job title");

    while(rs.next()) {

        String empNo= rs.getString("employeeNumber");

        String fname= rs.getString("firstname");

        String jTitle= rs.getString("jobTitle");

        System.out.printf("%s | %s | %s \n", empNo, fname, jTitle);

    }

    catch(Exception e){

        System.out.println(e.getMessage());

    }

}

}

```

---

#### **Line-by-line explanation (beginner friendly)**

##### **Imports**

1. import com.mysql.cj.jdbc.MysqlDataSource;

- **What:** Brings in the MysqDataSource class (MySQL's DataSource implementation).
- **Why:** You use MysqDataSource to obtain database connections in a modern, configurable way (instead of calling DriverManager directly). It supports better practices (e.g., can be used with connection pools later).

2. import java.io.FileInputStream;

- **What:** Lets you open and read a file as a stream of bytes.
- **Why:** You use it to read the db.properties file that stores your DB URL, user, and password.

3. import java.sql.Connection;

- **What:** The interface representing a connection (session) to the database.
- **Why:** You need a Connection object to run SQL statements.

4. import java.sql.DriverManager;

- **What:** A helper class that can give you DB connections (not used in your final connection flow since you use DataSource, but it's a common import).
- **Why:** Often used in simple examples; here MysqDataSource replaces the need for direct DriverManager usage.

5. import java.sql.ResultSet;

- **What:** Represents the result (rows) returned by a SELECT query.
- **Why:** You iterate over the ResultSet to read data returned from the database.

6. import java.sql.Statement;

- **What:** Used to run SQL statements (like SELECT, INSERT).
- **Why:** You create a Statement from the Connection to execute your query.

7. import java.util.Properties;

- **What:** A simple key-value map used to store configuration (strings).
- **Why:** You store and read dburl, user, password from an external file (db.properties) using Properties.

---

**Class and main method**

```
8. public class Main {
```

- **What:** Declares a class named Main. Java programs start from a main method inside a class.

```
9. public static void main(String[] args) {
```

- **What:** The program entry point — execution begins here.
- 

## Loading configuration

```
10. Properties prop = new Properties();
```

- **What:** Creates a Properties object to hold configuration values.
- **Why:** Keeps DB credentials outside the code (safer and easier to change).

```
11. try (FileInputStream fis = new FileInputStream("db.properties")) {
```

- **What:** Opens db.properties file **safely** using *try-with-resources*.
- **Why:** try-with-resources automatically closes the file even if an exception occurs — prevents resource leaks.

```
12. prop.load(fis);
```

- **What:** Reads key=value lines from db.properties into the prop object.
- **Why:** Now you can get values by keys like prop.getProperty("dburl").

```
13. } catch (Exception e) {
```

- **What:** Catches any error that happened while reading the file (file not found, permission denied, etc.).
- **Why:** Prevents the program from crashing and allows you to print a helpful message.

```
14. System.out.println("Error loading properties file: " + e.getMessage());
```

- **What:** Prints the error message.
  - **Why:** Helpful for debugging (you'll know why properties failed to load).
  - **Tip:** In real projects use e.printStackTrace() or a logger for full details.
- 

## Getting credentials from properties

```
15. String url = prop.getProperty("dburl");
```

- **What:** Reads the database URL (e.g., `jdbc:mysql://localhost:3306/yourdb`).
- **Why:** You need this to tell the driver which database to connect to.

```
16. String user = prop.getProperty("user");
```

```
17. String password = prop.getProperty("password");
```

- **What:** Read DB username and password.
- **Why:** Credentials to authenticate to the database.

```
18. String query = "SELECT * FROM employees";
```

- **What:** The SQL you want to run.
- **Why:** For demonstration — fetches all rows from the employees table.
- **Tip:** For user input or variables, prefer PreparedStatement to avoid SQL injection.

**Example db.properties file** (place in project root or give full path):

```
dburl=jdbc:mysql://localhost:3306/your_database_name
```

```
user=root
```

```
password=your_password
```

---

## Create and configure DataSource

```
19. MysqlDataSource datasource = new MysqlDataSource();
```

- **What:** Creates a MysqlDataSource object.
- **Why:** DataSource is a better way to get connections — cleaner and easier to manage (and can be wired to connection pools in real apps).

```
20. datasource.setURL(url);
```

```
21. datasource.setUser(user);
```

```
22. datasource.setPassword(password);
```

- **What:** Set the URL, username, and password into the DataSource.
- **Why:** Now the DataSource knows how to create authenticated connections to your DB.



## Open connection, execute query, read results (try-with-resources)

```
23. try(Connection conn = datasource.getConnection());
```

- **What:** Asks the DataSource for a Connection.
- **Why:** Connection is your session to the DB. Using *try-with-resources* ensures it is **automatically closed** when done.

```
24. Statement stmt = conn.createStatement();
```

- **What:** Creates a Statement object to execute SQL.
- **Why:** You need it to run executeQuery() or executeUpdate().

```
25. ResultSet rs = stmt.executeQuery(query)) {
```

- **What:** Executes the SELECT SQL and returns a ResultSet containing rows.
- **Why:** ResultSet lets you iterate over rows returned by the database.

```
26. System.out.println("Connected to database successfully!");
```

- **What:** Simple confirmation message.

```
27. System.out.println("Employee Number | Name | job title");
```

- **What:** Header for printed output.

```
28. while(rs.next()) {
```

- **What:** Moves the cursor to the next row; returns true while rows are available.
- **Why:** You loop through all rows returned by the query.

```
29. String empNo = rs.getString("employeeNumber");
```

```
30. String fname = rs.getString("firstname");
```

```
31. String jTitle = rs.getString("jobTitle");
```

- **What:** Reads column values from the current row by **column name**.
- **Why:** getString() returns the column value as a Java String. You can also use getInt(), getDate(), etc., depending on the column type.
- **Important:** Column names must match the table column names (case sensitivity depends on DB settings). You can also use column index: rs.getString(1).

```
32. System.out.printf("%s | %s | %s \n", empNo, fname, jTitle);
```

- **What:** Prints the row values in a formatted way.

- **Why:** Nicely displays each employee's data.

33. } catch(Exception e){

- **What:** Catches any exception in the DB block (connection failure, SQL error, etc.).

- **Why:** So the program doesn't crash and you can show a message or log the error.

34. System.out.println(e.getMessage());

- **What:** Prints the exception message.

- **Tip:** For debugging use e.printStackTrace() or a logger to see the full stack trace.
- 

## Important concepts & tips (so you really understand “why”)

- **Why use Properties and db.properties?**

Keeps credentials and URLs out of the code. Easier to change settings for different environments (dev/test/prod) and slightly safer than hardcoding.

- **Why MysqlDataSource instead of DriverManager?**

DataSource is a modern, flexible approach. It's easier to integrate with connection pooling (faster) and server-managed resources in real applications.

- **Why try-with-resources?**

try (resource) auto-closes resources (files, DB connections) — prevents resource leaks and is cleaner than manually closing in finally.

- **Why use ResultSet and rs.next()?**

The DB returns multiple rows; rs.next() lets you iterate row by row. Think of ResultSet as a cursor that points to the current row.

- **Why prefer PreparedStatement for dynamic queries?**

For queries with user input use PreparedStatement to safely pass parameters and avoid SQL injection, and it can be faster (precompiled).

- **Error handling:**

Printing e.getMessage() is okay for simple checks, but in real projects use e.printStackTrace() while developing and a proper logger in production.

---

## Small improvements you can make (recommended)

- Use PreparedStatement if your query needs parameters.
  - Use a logger (SLF4J/Logback) instead of System.out.println.
  - Don't store plain-text passwords in files — use environment variables or secret stores in production.
  - Validate that prop.getProperty(...) is not null before using it.
  - Close resources in finally (or keep using try-with-resources as you do — good!).
- 

### Short checklist to run this program successfully

1. Create db.properties file in the project root with keys dburl, user, password.
  2. Add the MySQL JDBC driver JAR (e.g., mysql-connector-j-8.x.x.jar) to your project classpath.
  3. Ensure MySQL server is running and employees table exists with columns employeeNumber, firstname, jobTitle.
  4. Run the program — it will print rows from the employees table.
- 

If you want, I can:

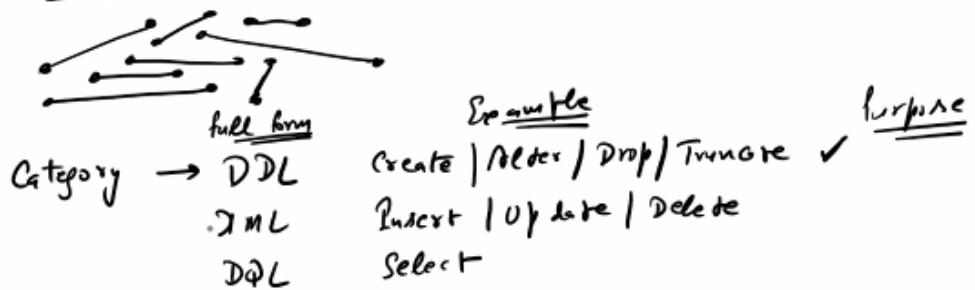
- show a **sample db.properties** file ready to paste,
- convert your SELECT query into a **PreparedStatement** example, or
- show how to **handle null columns** and different SQL types (int, date, etc.).

Which would you like next?

Excellent question, Nandhini 🌸 — when we talk about **languages in SQL**, we mean the **different types of commands** or **categories** of SQL statements that perform various kinds of tasks in a database.

Let's understand this clearly 🤔

## Crud Operation

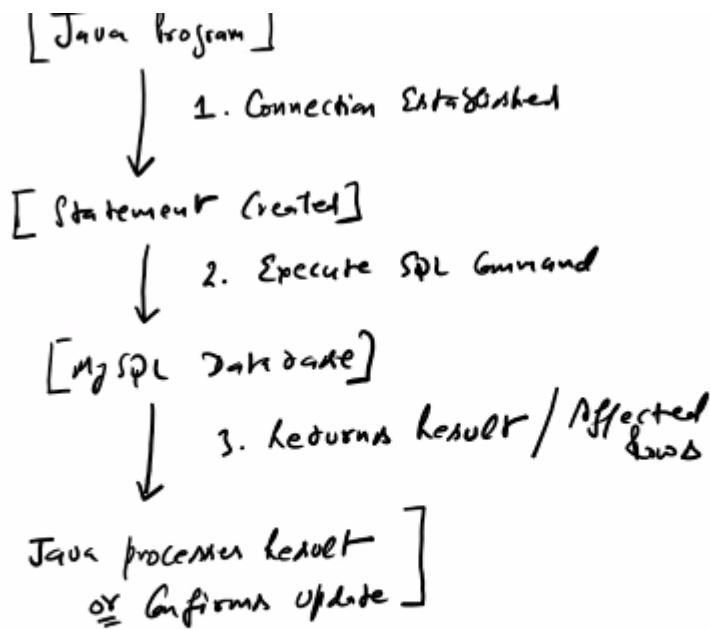


DDL → Architect

DML → Worker

DQL → Reporter

| <u>Method</u>   | <u>Returns</u>                | <u>Usage</u>             |
|-----------------|-------------------------------|--------------------------|
| executeQuery()  | ResultSet                     | SQL (Select)             |
| executeUpdate() | int<br><u>(rows affected)</u> | DML / DDL                |
| execute()       | <u>Boolean</u>                | Unknown / Mixed Commands |



## Types of SQL Languages

SQL is divided into **five main categories**, based on the kind of operation they perform:

| Type       | Full Form                    | Purpose                                                               | Examples                      |
|------------|------------------------------|-----------------------------------------------------------------------|-------------------------------|
| <b>DDL</b> | Data Definition Language     | Defines and manages <b>database structure</b> (tables, schemas, etc.) | CREATE, ALTER, DROP, TRUNCATE |
| <b>DML</b> | Data Manipulation Language   | Manages <b>data inside tables</b> (insert, update, delete)            | INSERT, UPDATE, DELETE        |
| <b>DQL</b> | Data Query Language          | Used to <b>query (read)</b> data from tables                          | SELECT                        |
| <b>TCL</b> | Transaction Control Language | Manages <b>transactions</b> (group of SQL operations)                 | COMMIT, ROLLBACK, SAVEPOINT   |
| <b>DCL</b> | Data Control Language        | Controls <b>access and permissions</b> to database objects            | GRANT, REVOKE                 |

## 1 DDL (Data Definition Language)

- 💡 Used to define or change **structure** of the database — tables, columns, indexes, etc.

| Command Description |                                         | Example                                           |
|---------------------|-----------------------------------------|---------------------------------------------------|
| CREATE              | Creates new table or database           | CREATE TABLE students (id INT, name VARCHAR(50)); |
| ALTER               | Modifies table structure                | ALTER TABLE students ADD age INT;                 |
| DROP                | Deletes a table or database completely  | DROP TABLE students;                              |
| TRUNCATE            | Deletes all records but keeps structure | TRUNCATE TABLE students;                          |

- 💡 DDL changes are **auto-committed**, meaning they cannot be rolled back.
- 

## ✳️ 2 DML (Data Manipulation Language)

- 💡 Used to **manipulate data** inside existing tables.

| Command Description |                      | Example                                      |
|---------------------|----------------------|----------------------------------------------|
| INSERT              | Add new records      | INSERT INTO students VALUES (1, 'Nandhini'); |
| UPDATE              | Modify existing data | UPDATE students SET name='Priya' WHERE id=1; |
| DELETE              | Remove records       | DELETE FROM students WHERE id=1;             |

- 💡 DML changes **can be rolled back** (if using transactions).
- 

## ✳️ 3 DQL (Data Query Language)

- 💡 Used to **retrieve data** from the database.

| Command Description |                            | Example                 |
|---------------------|----------------------------|-------------------------|
| SELECT              | Fetch records from a table | SELECT * FROM students; |

- 💡 SELECT is the only DQL command — it just **reads** data.
- 

## ✳️ 4 TCL (Transaction Control Language)

 Used to manage the **changes made by DML** commands.

| Command   | Description                       | Example           |
|-----------|-----------------------------------|-------------------|
| COMMIT    | Save all changes permanently      | COMMIT;           |
| ROLLBACK  | Undo changes (before commit)      | ROLLBACK;         |
| SAVEPOINT | Create a temporary rollback point | SAVEPOINT point1; |

 TCL ensures **data integrity**— all or none of a transaction happens.

---

## **5 DCL (Data Control Language)**

 Used to **control access and permissions** for database users.

| Command | Description             | Example                               |
|---------|-------------------------|---------------------------------------|
| GRANT   | Give permission to user | GRANT SELECT ON students TO user1;    |
| REVOKE  | Take back permission    | REVOKE SELECT ON students FROM user1; |

 DCL is used mostly by **database administrators (DBAs)**.

---

## **Summary Table**

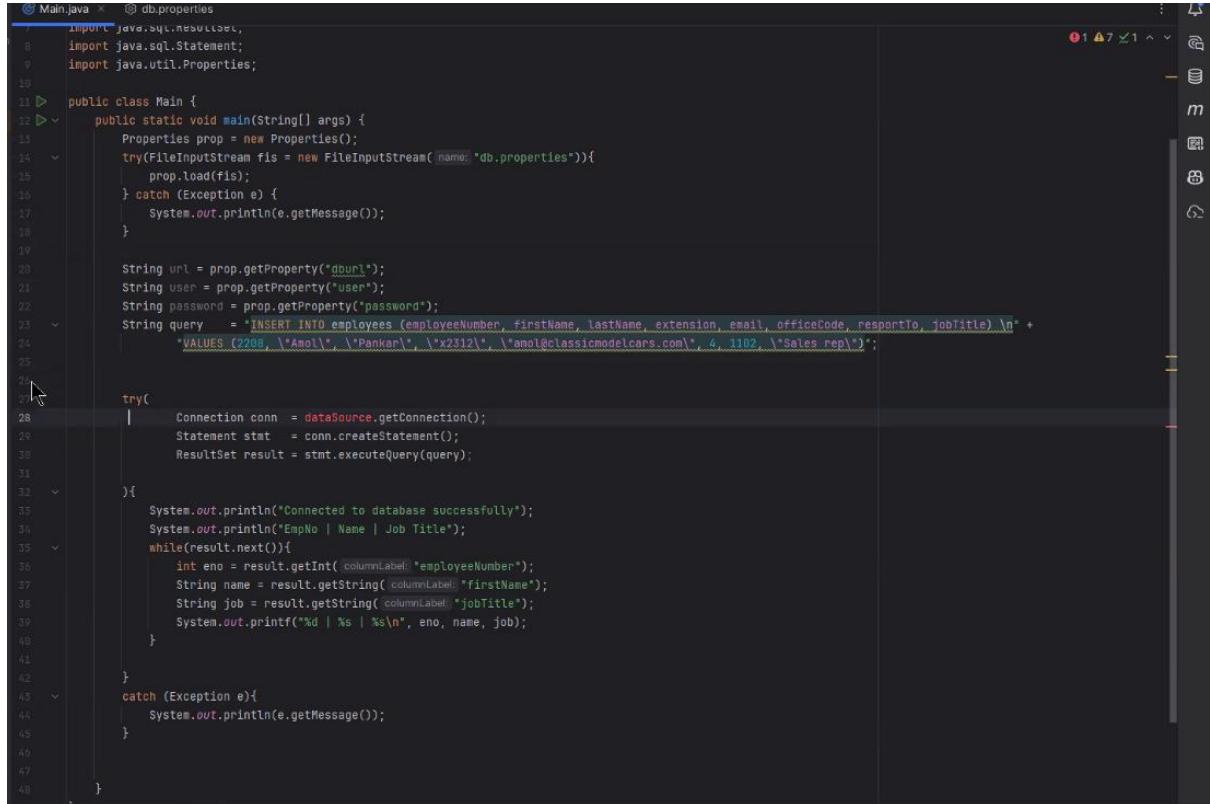
| Language Purpose | Example Commands     |                               |
|------------------|----------------------|-------------------------------|
| DDL              | Define structure     | CREATE, ALTER, DROP, TRUNCATE |
| DML              | Manage data          | INSERT, UPDATE, DELETE        |
| DQL              | Query data           | SELECT                        |
| TCL              | Control transactions | COMMIT, ROLLBACK, SAVEPOINT   |
| DCL              | Manage permissions   | GRANT, REVOKE                 |

## **In Short**

- **DDL** → Defines structure
- **DML** → Works with data
- **DQL** → Reads data

- **TCL** → Controls transactions
  - **DCL** → Manages access
- 

Would you like me to give a **real-time example** combining all 5 SQL language types (like a mini database workflow)?



```

1  Main.java x  db.properties
2  import java.sql.Connection;
3  import java.sql.DriverManager;
4  import java.sql.Statement;
5  import java.util.Properties;
6
7  public class Main {
8      public static void main(String[] args) {
9          Properties prop = new Properties();
10         try(FileInputStream fis = new FileInputStream("name: db.properties")){
11             prop.load(fis);
12         } catch (Exception e) {
13             System.out.println(e.getMessage());
14         }
15
16         String url = prop.getProperty("dburl");
17         String user = prop.getProperty("user");
18         String password = prop.getProperty("password");
19         String query = "INSERT INTO employees (employeeNumber, firstName, lastName, extension, email, officeCode, reportTo, jobTitle) \n" +
20             "VALUES (200, 'Amol', 'Pankar', 'x2312', 'amol@classicmodelcars.com', 4, 1102, 'Sales rep')";
21
22         try{
23             Connection conn = dataSource.getConnection();
24             Statement stat = conn.createStatement();
25             ResultSet result = stat.executeQuery(query);
26
27             System.out.println("Connected to database successfully");
28             System.out.println("EmpNo | Name | Job Title");
29             while(result.next()){
30                 int eno = result.getInt(columnLabel("employeeNumber"));
31                 String name = result.getString(columnLabel("firstName"));
32                 String job = result.getString(columnLabel("jobTitle"));
33                 System.out.printf("%d | %s | %s\n", eno, name, job);
34             }
35         }
36         catch (Exception e){
37             System.out.println(e.getMessage());
38         }
39     }
40 }
41
42
43
44
45
46
47
48

```

```

Main.java x  @@ db.properties
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.ResultSet;
4 import java.sql.Statement;
5 import java.util.Properties;
6
7 public class Main {
8     public static void main(String[] args) {
9         Properties prop = new Properties();
10        try(FileInputStream fis = new FileInputStream(new File("db.properties"))){
11            prop.load(fis);
12        } catch (Exception e) {
13            System.out.println(e.getMessage());
14        }
15
16        String url = prop.getProperty("dburl");
17        String user = prop.getProperty("user");
18        String password = prop.getProperty("password");
19        String query = "INSERT INTO employees (employeeNumber, firstName, lastName, extension, email, officeCode, reportsTo, jobTitle) \n" +
20                    "VALUES (2208, 'Amol', 'Pankar', 'x2312', 'amol@classicmodelcars.com', 4, 1102, 'Sales rep')";
21
22
23        try{
24            Connection conn = DriverManager.getConnection(url, user, password);
25            Statement stmt = conn.createStatement();
26
27        }{
28            int rows = stmt.executeUpdate(query);
29            System.out.println(rows + " record(s) inserted successfully.");
30        }
31
32        catch (Exception e){
33            System.out.println(e.getMessage());
34        }
35
36    }
37
38
39
40
41
42

```

The screenshot shows the MySQL Workbench interface. In the top navigation bar, 'Local Instance MySQL80' is selected. The 'SQL' tab is active, displaying the following SQL code:

```

1 • use learningmysql;
2 • SELECT * FROM employees;

```

In the 'Navigator' pane, the 'Schemas' section is expanded, showing the 'learningmysql' schema with its tables: employees, customers, offices, orderdetails, parts, payments, products, regions, and regions\_products. The 'employees' table is selected.

The main workspace displays the 'Result Grid' for the 'employees' table, showing 22 rows of data. The columns are: employeeNumber, lastName, firstName, extension, email, officeCode, reportsTo, and jobTitle. The data includes entries like Amol Pankar, George Lou, and others.

At the bottom, the 'Output' pane shows the execution history and messages:

- 1 15:09:26 use learningmysql
- 2 15:09:50 SELECT \* FROM employees LIMIT 0, 1000
- 3 17:23:17 SELECT \* FROM learningmysql\_tempseconderies LIMIT 0, 1000
- 4 17:24:14 SELECT \* FROM learningmysql\_employees LIMIT 0, 1000
- 5 17:35:33 SELECT \* FROM employees LIMIT 0, 1000

Messages include: 0 rows affected, 23 rows returned, 12 rows returned, 23 rows returned, and 24 rows returned.

```

import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Properties;

```

```

public class Main {

```

```

public static void main(String[] args) {
    Properties prop = new Properties();

    // Step 1: Load db.properties file
    try (FileInputStream fis = new FileInputStream("db.properties")) {
        prop.load(fis);
    } catch (Exception e) {
        System.out.println("Error loading properties file: " + e.getMessage());
        return; // stop execution if properties not loaded
    }

    // Step 2: Read database credentials
    String url = prop.getProperty("dburl");
    String user = prop.getProperty("user");
    String password = prop.getProperty("password");

    // Step 3: Create SQL query to insert data
    String query = "INSERT INTO employees (employeeNumber, firstName, lastName,
extension, email, officeCode, reportsTo, jobTitle) " +
        "VALUES (2208, 'nandhini', 'srinivasan', 'x2312', 'amol@classicmodelcars.com',
4, 1102, 'Sales Rep')";

    // Step 4: Connect to database and execute query
    try (Connection conn = DriverManager.getConnection(url, user, password);
        Statement stmt = conn.createStatement()) {

        int rows = stmt.executeUpdate(query);
        System.out.println(rows + " record(s) inserted successfully!");

    } catch (Exception e) {
        System.out.println("Database error: " + e.getMessage());
    }
}

```

Perfect, Nandhini 🌸 — this is a **great JDBC example** that performs an **INSERT operation** into a MySQL database using a **properties file** for configuration.

Let's go **line-by-line** and understand what each part means — from scratch, assuming you're completely new 👋

---

## Step-by-Step Explanation

---

```java

```
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.util.Properties;
```

### \*\*What this does:\*\*

These are **import statements** — they bring classes from Java libraries into your program so you can use them.

- `java.io.FileInputStream` → used to **read a file** (in this case, the `db.properties` file).
- `java.sql.Connection` → represents a **connection to your database**.
- `java.sql.DriverManager` → used to **connect** Java with your database (like MySQL).
- `java.sql.Statement` → used to **execute SQL queries** (like `INSERT`, `UPDATE`, etc.).
- `java.util.Properties` → used to **read key-value pairs** (like username, password) from a ` `.properties` file.

---

```
### ``java
public class Main {
    public static void main(String[] args) {
```

### What this does:

- This defines your **main class** Main.
- Inside it, `public static void main(String[] args)` is the **entry point** of your Java program — execution starts here.

---

```
```java
```

```
Properties prop = new Properties();
```

 \*\*What this does:\*\*

- Creates a new `Properties` object.
- Think of it as a small \*\*dictionary\*\* that stores settings in key-value format (e.g., `user=root`, `password=1234`).

---

```
### `` java
```

```
try (FileInputStream fis = new FileInputStream("db.properties")) {  
    prop.load(fis);  
}  
catch (Exception e) {  
    System.out.println("Error loading properties file: " + e.getMessage());  
    return;  
}
```

 **Explanation:**

- `FileInputStream("db.properties")` → opens the file named **db.properties** (where you've stored your DB credentials).
- `prop.load(fis);` → loads the contents of that file into your `Properties` object.
- The code is inside a **try-with-resources** block → this automatically **closes** the file after reading (no memory leaks).
- `catch (Exception e)` → catches any error (e.g., file missing, wrong path) and prints an error message.
- `return;` → stops the program if the file isn't loaded successfully.

 **Example of your db.properties file:**

```
dburl=jdbc:mysql://localhost:3306/classicmodels
```

```
user=root  
password=1234
```

---

```
```java  
String url = prop.getProperty("dburl");  
String user = prop.getProperty("user");  
String password = prop.getProperty("password");
```

 \*\*What this does:\*\*

- Reads values from the properties file.
- Example:
  - `url` → `jdbc:mysql://localhost:3306/classicmodels`
  - `user` → `root`
  - `password` → `1234`
- This helps you \*\*avoid hardcoding credentials\*\* directly inside Java (which is a good security practice).

---

```
### `` `java
```

```
String query = "INSERT INTO employees (employeeNumber, firstName, lastName,  
extension, email, officeCode, reportsTo, jobTitle) " +  
    "VALUES (2208, 'nandhini', 'srinivasan', 'x2312', 'amol@classicmodelcars.com', 4,  
1102, 'Sales Rep')";
```

 **What this does:**

- This is your **SQL query** — written as a Java String.
- It will insert one new record into the employees table.
- The + allows you to **split the string** across lines for better readability.

 Example equivalent SQL (executed inside MySQL):

```
INSERT INTO employees (employeeNumber, firstName, lastName, extension, email, officeCode, reportsTo, jobTitle)
```

```
VALUES (2208, 'nandhini', 'srinivasan', 'x2312', 'amol@classicmodelcars.com', 4, 1102, 'Sales Rep');
```

---

```
```java
```

```
try (Connection conn = DriverManager.getConnection(url, user, password);  
Statement stmt = conn.createStatement()) {
```

 \*\*Explanation:\*\*

- This line \*\*connects your Java program to the MySQL database\*\*.

Breakdown:

- `DriverManager.getConnection(url, user, password)` → connects to your database using the info from `db.properties`.
- Returns a `Connection` object named `conn`.
- `conn.createStatement()` → creates a `Statement` object named `stmt` to \*\*run SQL commands\*\*.
- The `try (...)` again is a \*\*try-with-resources\*\* — both the connection and statement will \*\*close automatically\*\* after use (good practice).

---

```
### ``java
```

```
int rows = stmt.executeUpdate(query);  
System.out.println(rows + " record(s) inserted successfully!");
```

 **Explanation:**

- `stmt.executeUpdate(query)` → runs the **INSERT** SQL command.
- Returns the number of rows affected (in this case, 1 if insertion is successful).
- The message prints something like:

- 1 record(s) inserted successfully!
- 

```java

```
} catch (Exception e) {  
    System.out.println("Database error: " + e.getMessage());  
}
```

 \*\*Explanation:\*\*

- If anything goes wrong (e.g., wrong SQL syntax, connection issue, duplicate primary key), it prints the error message.
  - This helps in debugging.
- 

##  \*\*In Simple Words\*\*

| Step                              | What Happens                                   | Why It's Needed |
|-----------------------------------|--|-----------------|
| 1   Load the `db.properties` file | To get database details safely                 |                 |
| 2   Read URL, user, password      | So we can connect to MySQL                     |                 |
| 3   Write the SQL query           | To insert a new record                         |                 |
| 4   Connect using `DriverManager` | Establish communication between Java and MySQL |                 |
| 5   Execute SQL using `Statement` | Run the query on the database                  |                 |
| 6   Display success message       | Confirm the operation                          |                 |
| 7   Catch and print errors        | Handle problems gracefully                     |                 |

---

###  \*\*How this looks in real life\*\*

Imagine you're filling a form to add a new employee →

Java collects your input → builds an SQL query → sends it to MySQL → MySQL stores it in the `employees` table.

---

Would you like me to modify this same code to include \*\*reading user input (like employee name, job title)\*\* dynamically from the console instead of hardcoding values? That's often the next step in JDBC learning.

Perfect  Let's now learn how to **modify (update) data** in SQL using **JDBC (Java Database Connectivity)** — step by step with complete explanation.

---

### Concept: What is “Modify” in SQL?

In SQL, **modifying data** means changing the existing records inside a table.

The SQL command used is:

UPDATE table\_name

SET column1 = new\_value, column2 = new\_value

WHERE condition;

The WHERE condition ensures **only specific rows** are updated (otherwise, all rows will change!).

---

### Example: Java program to modify (update) employee data

```
import java.io.FileInputStream;  
  
import java.sql.Connection;  
  
import java.sql.DriverManager;  
  
import java.sql.Statement;  
  
import java.util.Properties;
```

```
public class UpdateData {  
    public static void main(String[] args) {  
        Properties prop = new Properties();  
  
        // Step 1: Load db.properties file (for credentials)  
        try (FileInputStream fis = new FileInputStream("db.properties")) {  
            prop.load(fis);  
        } catch (Exception e) {  
            System.out.println("Error loading properties file: " + e.getMessage());  
            return;  
        }  
  
        // Step 2: Read database credentials from properties file  
        String url = prop.getProperty("dburl");  
        String user = prop.getProperty("user");  
        String password = prop.getProperty("password");  
  
        // Step 3: Write SQL query to update employee details  
        String query = "UPDATE employees SET jobTitle = 'Sales Manager' WHERE  
employeeNumber = 2208";  
  
        // Step 4: Connect to database and execute update query  
        try (Connection conn = DriverManager.getConnection(url, user, password);  
             Statement stmt = conn.createStatement()) {  
  
            int rows = stmt.executeUpdate(query);  
            System.out.println(rows + " record(s) updated successfully!");  
        }  
    }  
}
```

```

} catch (Exception e) {
    System.out.println("Database error: " + e.getMessage());
}
}

}

```

---

## Explanation — Line by Line

| Line / Section  | Explanation  |
|---|--|
| import java.io.FileInputStream;   | To read external files (like db.properties) containing credentials.              |
| import java.sql.*;  | Imports all SQL classes — Connection, DriverManager, Statement.                  |
| import java.util.Properties;  | Used to handle key-value pairs (like dburl, user, password).                     |
| Properties prop = new Properties();   | Creates a Properties object to load and store database settings.                 |
| try (FileInputStream fis = new<br>FileInputStream("db.properties")) {<br>prop.load(fis); }          | Opens and reads the db.properties file safely. If it fails, the error is caught. |
| String url = prop.getProperty("dburl");   | Reads the database URL (like jdbc:mysql://localhost:3306/classicmodels).         |
| String user = prop.getProperty("user");   | Reads the username for login.  |
| String password =<br>prop.getProperty("password");  | Reads the password for login.  |
| String query = "UPDATE employees SET<br>jobTitle = 'Sales Manager' WHERE<br>employeeNumber = 2208"; | The actual <b>SQL update command</b> — modifies job title of employee 2208.      |

| Line / Section  | Explanation   |
|---|---|
| Connection conn =<br>DriverManager.getConnection(url,<br>user, password); | Establishes connection between Java and MySQL database.         |
| Statement stmt =<br>conn.createStatement();                               | Creates a Statement object to send the SQL query to the DB.     |
| int rows = stmt.executeUpdate(query);                                     | Executes the SQL update and returns how many rows were changed. |
| System.out.println(rows + " record(s)<br>updated successfully!");         | Displays success message showing how many rows were updated.    |
| catch (Exception e)   | Catches any database or connection errors.                      |

---

### Example Output

1 record(s) updated successfully!

This means **one row was successfully modified** in the table.

---

### Summary

| Operation              | SQL Command               | Java Method            | Purpose                        |
|------------------------|---------------------------|------------------------|--------------------------------|
| Create                 | INSERT INTO ...           | executeUpdate()        | Add a new record               |
| Read                   | SELECT * FROM ...         | executeQuery()         | Retrieve records               |
| <b>Update (Modify)</b> | <b>UPDATE ... SET ...</b> | <b>executeUpdate()</b> | <b>Change existing records</b> |
| Delete                 | DELETE FROM ...           | executeUpdate()        | Remove records                 |

---

Would you like me to show the “**Delete data**” example next (similar step-by-step explanation)?

```

Main.java  db.properties

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class Main {
    public static void main(String[] args) {
        Properties prop = new Properties();
        try(FileInputStream fis = new FileInputStream("name: db.properties")){
            prop.load(fis);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        String url = prop.getProperty("url");
        String user = prop.getProperty("user");
        String password = prop.getProperty("password");
        String query = "INSERT INTO employees (employeeNumber, firstName, lastName, extension, email, officeCode, reportsTo, jobTitle) \n" +
                       "VALUES (2333, 'Anil', 'Pankar', 'x2312', 'anil@classicalmodelcars.com', 4, 1102, 'Sales rep')";

        try{
            Connection conn = DriverManager.getConnection(url, user, password);
            Statement stat = conn.createStatement();

        }{
            int rows = stat.executeUpdate(query);
            System.out.println(rows + " records inserted successfully.");
        }
        catch (Exception e){
            System.out.println(e.getMessage());
        }

    }
}

```

MySQL Workbench

Local Instance MySQL >

File Edit View Query Database Server Tools Scripting Help

Navigator: SCHEMAS learningmysql

Tables: employees

Columns: employeeNumber, lastName, firstName, extension, email, officeCode, reportsTo, jobTitle

Result Grid:

| employeeNumber | lastName  | firstName | extension | email                             | officeCode | reportsTo | jobTitle  |
|----------------|-----------|-----------|-----------|-----------------------------------|------------|-----------|-----------|
| 1333           | Voulgaris | George    | x1002     | gvoulgaris@classicalmodelcars.com | 3          | 1143      | Sales Rep |
| 1337           | Bandur    | Lou       | x6493     | lbandur@classicalmodelcars.com    | 4          | 1102      | Sales Rep |
| 1370           | Hernandez | Gerard    | x2038     | ghernandez@classicalmodelcars.com | 4          | 1102      | Sales Rep |
| 1401           | Castillo  | Pamela    | x2799     | pcastillo@classicalmodelcars.com  | 4          | 1102      | Sales Rep |
| 1501           | Butt      | Larry     | x2311     | lbutt@classicalmodelcars.com      | 7          | 1102      | Sales Rep |
| 1504           | Jones     | Berry     | x302      | bjones@classicalmodelcars.com     | 7          | 1102      | Sales Rep |
| 1611           | Peter     | Andy      | x101      | apeter@classicalmodelcars.com     | 6          | 1088      | Sales Rep |
| 1612           | Martini   | Peter     | x302      | pmartini@classicalmodelcars.com   | 6          | 1088      | Sales Rep |
| 1619           | King      | Tom       | x303      | tking@classicalmodelcars.com      | 6          | 1088      | Sales Rep |
| 1621           | Nieto     | Plant     | x201      | rniesto@classicalmodelcars.com    | 5          | 1096      | Sales Rep |
| 1625           | Kemp      | Yvonne    | x302      | ykemp@classicalmodelcars.com      | 5          | 1021      | Sales Rep |
| 1702           | Gerard    | Martin    | x2112     | mg Gerard@classicalmodelcars.com  | 4          | 1102      | Sales Rep |
| 2208           | Pankar    | Anil      | x2312     | anil@classicalmodelcars.com       | 4          | 1102      | Sales Rep |

Action Output:

- 1 15:09:26 use learningmysql
- 2 15:09:30 SELECT \* FROM employees LIMIT 0, 1000
- 3 17:23:17 SELECT \* FROM learningmysql.tensecondorders LIMIT 0, 1000
- 4 17:24:14 SELECT \* FROM learningmysql.employees LIMIT 0, 1000
- 5 17:36:33 SELECT \* FROM employees LIMIT 0, 1000
- 6 17:36:36 SELECT \* FROM employees LIMIT 0, 1000
- 7 17:49:46 SELECT \* FROM employees LIMIT 0, 1000

```
// Defines the package name for organizing your Java files

import java.io.FileInputStream; // To read the db.properties file
import java.sql.Connection; // Represents a connection to the database
import java.sql.DriverManager; // Used to get database connections
import java.sql.Statement; // Used to execute SQL statements
import java.util.Properties; // To load key-value pairs from a properties file

public class Main {
    public static void main(String[] args) {

        // Step 1: Create a Properties object to hold database connection info
        Properties prop = new Properties();

        // Step 2: Load db.properties file using try-with-resources
        // try-with-resources automatically closes the FileInputStream after use
        try (FileInputStream fis = new FileInputStream("db.properties")) {
            // Load the properties (key=value pairs) from the file
            prop.load(fis);
        } catch (Exception e) {
            // If there's an error (e.g., file not found), print error message
            System.out.println("Error loading properties file: " + e.getMessage());
            return; // Exit the program if properties file not loaded properly
        }

        // Step 3: Read database connection details from the properties file
        // The file should contain keys like dburl, user, and password
        String url = prop.getProperty("dburl"); // e.g.
        jdbc:mysql://localhost:3306/classicmodels
        String user = prop.getProperty("user"); // e.g. root
        String password = prop.getProperty("password"); // e.g. 1234

        // Step 4: Write the SQL INSERT query to add a new employee record
        String query = "UPDATE employees SET jobTitle='Junior Developer' WHERE
employeeNumber=2208";

        // Step 5: Establish the connection and execute the SQL statement
        // try-with-resources ensures the connection and statement are closed
        automatically
```

```

try (Connection conn = DriverManager.getConnection(url, user, password); // Open
connection
      Statement stmt = conn.createStatement()) { // Create statement object to send
SQL commands

    // Step 6: Execute the INSERT query
    // executeUpdate() is used for INSERT, UPDATE, or DELETE queries
    int rows = stmt.executeUpdate(query);

    // Step 7: Print confirmation of how many records were inserted
    System.out.println(rows + " record(s) updated successfully!");

} catch (Exception e) {
    // If any SQL or connection error occurs, print the error message
    System.out.println("Database error: " + e.getMessage());
}
}
}
}

```

Entire code of oct 28

```

//// Package declaration - this defines the package name for your class
//package com.example;
//
//// Importing the necessary JDBC classes
//import java.sql.Connection; // Used to create a connection with the database
//import java.sql.DriverManager; // Used to manage JDBC drivers and establish
connection
//import java.sql.SQLException; // Used to handle SQL-related exceptions
//
//public class Main{
//  public static void main(String[] args){
//
//    // Database credentials (username and password)
//    String user = "root"; // MySQL username
//    String pass = "12345"; // MySQL password
//
//    // URL to connect to the MySQL database
//    // jdbc:mysql:// - protocol for MySQL database
//    // localhost - means the database is running on your local computer
//    // 3306 - default MySQL port number
//    // learningmysql - name of the database you created in MySQL

```

```
// String url = "jdbc:mysql://localhost:3306/learningmysql";
//
// System.out.println("Connecting to the database...");
//
// // Step 1: Declare the connection variable
// // Initially set to null (no connection yet)
// Connection conn = null;
//
// try {
//
//
// // Step 3: Establish connection to the database
// // DriverManager tries to connect using the given URL, username, and password.
// conn = DriverManager.getConnection(url, user, pass);
//
// // If connection succeeds, this line executes
// System.out.println(" Connected to the database!");
//
// } catch (Exception e) {
// // Step 4: If any error occurs (e.g., wrong password or driver issue)
// // the catch block will execute
// System.out.println(" Error: " + e.getMessage());
// } finally {
// // Step 5: This block runs no matter what happens (success or failure)
// // Used to safely close the connection to avoid memory leaks
// if (conn != null) {
// try {
// conn.close(); // Close the connection when done
// System.out.println(" Connection closed.");
// } catch (SQLException e) {
// // Handle any errors that occur while closing
// System.out.println(" Error while closing: " + e.getMessage());
// }
// }
// }
// }
```

```
//package com.example;
//
//import java.sql.Connection;
//import java.sql.DriverManager;
//import java.sql.SQLException;
//
//public class Main {
//    public static void main(String[] args) {
//        String user = "root";
//        String pass = "12345";
//        String url = "jdbc:mysql://localhost:3306/learningmysql";
//
//        System.out.println("Connecting to the database...");
//        Connection conn = null;
//
//        try {
//            // Establish connection
//            conn = DriverManager.getConnection(url, user, pass);
//            System.out.println(" Connected to the database!");
//
//        } catch (Exception e) {
//            System.out.println(" Error: " + e.getMessage());
//        }
//        finally {
//            // Close connection safely
//            if (conn != null) {
//                try {
//                    conn.close();
//                    System.out.println(" Connection closed.");
//                } catch (SQLException e) {
//                    System.out.println(" Error while closing: " + e.getMessage());
//                }
//            }
//        }
//    }
//}
```

// another method

//• Uses Java 7+ feature: Try-with-Resources.

```
//      • Automatically closes the connection after the try block, even if an exception occurs.  
//      • Cleaner, safer, and more readable.  
//      • No need for a finally block or manual close.
```

```
package com.example;  
//import java.sql.Connection;  
//import java.sql.DriverManager;  
//import java.sql.SQLException;  
  
//  
//public class Main {  
//    public static void main(String[] args) {  
//        String user = "root";  
//        String pass = "12345";  
//        String url = "jdbc:mysql://localhost:3306/learningmysql"; // ✅ Correct URL  
//  
//        System.out.println("Connecting to the database...");  
//  
//  
//        try( Connection conn = DriverManager.getConnection(url, user, pass)) {  
//  
//            // Establish connection  
//            if(conn != null){  
//                System.out.println(" Connected to the database!");  
//            }  
//            } catch (Exception e) {  
//                System.out.println("Connection Failed... ");  
//            }  
//        }  
//    }  
// another method mordern way  
// try with idea way( try with conn)  
  
//import java.sql.Connection;  
//import java.sql.DriverManager;  
//import java.sql.SQLException;  
  
//  
//public class Main {  
//    public static void main(String[] args) {
```

```
// String user = "root";
// String pass = "12345564";
// String url = "jdbc:mysql://localhost:3306/learningmysql"; // ✅ Correct URL
//
// System.out.println("Connecting to the database...");
//
//
// try{
//     Connection conn = DriverManager.getConnection(url, user, pass);
//     // Establish connection
//     if(conn != null){
//         System.out.println(" Connected to the database!");
//     }
// } catch (Exception e){
//     System.out.println("Connection Failed... ");
// }
// }

//package com.example; // Package name for organizing your code
//
//// Importing required JDBC classes
//import java.sql.Connection; // Used to establish a connection with the database
//import java.sql.DriverManager; // Used to get the database connection
//import java.sql.SQLException; // Handles SQL-related exceptions
//
//public class Main {
//    public static void main(String[] args) {
//
//        // Database credentials and URL
//        String user = "root"; // MySQL username
//        String pass = "12345"; // MySQL password
//        String url = "jdbc:mysql://localhost:3306/learningmysql";
//        // ↑ JDBC URL format: jdbc:mysql://<host>:<port>/<database_name>
//
//        System.out.println("Connecting to the database...");
//
//        // try-with-resources — automatically closes the connection after try block ends
//        try (Connection conn = DriverManager.getConnection(url, user, pass)) {
//
//            // Check if the connection is successfully established
//        }
//    }
//}
```

```
//      if (conn != null) {
//          System.out.println(" Connected to the database!");
//      }
//
//  } catch (SQLException e) {
//      // Handles any SQL or connection errors
//      System.out.println(" Connection Failed! Error: " + e.getMessage());
//  }
//
//  // No need for 'finally' block — connection closes automatically here
// }
//}

//import java.sql.Connection;
//import java.sql.DriverManager;
//
//public class Main {
//    public static void main(String[] args) {
//        String user = "root";
//        String password = "12345";
//        String url = "jdbc:mysql://localhost:3306/learningmysql";
//
//        System.out.println("Connecting to the database...");
//
//        try{
//            // 1. Load the JDBC driver
//            Class.forName("com.mysql.cj.jdbc.Driver");
//
//            // 2. Establish the connection
//            Connection conn = DriverManager.getConnection(url, user, password);
//            System.out.println("Connection established successfully!");
//
//            // 3. Close the connection
//            conn.close();
//        } catch (Exception e){
//            System.out.println(e.getMessage());
//        }
//    }
//}

//import java.io.FileInputStream;
//import java.sql.Connection;
//import java.sql.DriverManager;
```

```
//import java.util.Properties;
//
//public class Main {
//    public static void main(String[] args) {
//        //
//        System.out.println("Connecting to the database...");
//        //
//        try {
//            // 1 Load properties from external file
//            Properties prop = new Properties();
//            prop.load(new FileInputStream("db.properties"));
//            //
//            // 2 Read values from db.properties file
//            String url = prop.getProperty("dburl");
//            String user = prop.getProperty("user");
//            String password = prop.getProperty("password");
//            //
//            // 3 Load the JDBC driver manually
//            Class.forName("com.mysql.cj.jdbc.Driver");
//            //
//            // 4 Establish the connection
//            Connection conn = DriverManager.getConnection(url, user, password);
//            System.out.println("Connection established successfully!");
//            //
//            // 5 Close the connection
//            conn.close();
//            //
//        } catch (Exception e) {
//            System.out.println(e.getMessage());
//        }
//    }
//}
```

//The first code loads the JDBC driver manually

```
//    using Class.forName() and doesn't use try-with-resources,
//    so connections and streams must be closed manually.
//    The second code uses try-with-resources for safer resource
//    handling and automatically closes the FileInputStream and Connection,
//    removing the need to load the driver explicitly.
```

```
//import java.io.FileInputStream;
//import java.sql.Connection;
//import java.sql.DriverManager;
```

```
//import java.util.Properties;
//
//public class Main {
//    public static void main(String[] args) {
//        System.out.println("Connecting to the database...");
//
//        // Step 1: Create a Properties object
//        Properties prop = new Properties();
//
//        // Step 2: Load db.properties file safely using try-with-resources
//        try (FileInputStream fis = new FileInputStream("db.properties")) {
//            prop.load(fis);
//        } catch (Exception e) {
//            System.out.println("Error loading properties file: " + e.getMessage());
//            return; // Exit if properties not loaded
//        }
//
//        // Step 3: Get database credentials from file
//        String url = prop.getProperty("dburl");
//        String user = prop.getProperty("user");
//        String password = prop.getProperty("password");
//
//        // Step 4: Try to connect to the database
//        try (Connection conn = DriverManager.getConnection(url, user, password)) {
//            if (conn != null) {
//                System.out.println(" Connected to the database successfully!");
//            } else {
//                System.out.println(" Connection object is null!");
//            }
//        } catch (Exception e) {
//            System.out.println("Connection Failed: " + e.getMessage());
//        }
//    }
//}
//
//import java.io.FileInputStream;
//import java.sql.Connection;
//import java.sql.DriverManager;
//import java.sql.ResultSet;
//import java.sql.Statement;
//import java.util.Properties;
//
```

```
//public class Main {  
//    public static void main(String[] args) {  
//  
//        System.out.println("Connecting to the database...");  
//  
//        // Step 1: Create a Properties object  
//        Properties prop = new Properties();  
//  
//        // Step 2: Load db.properties file safely using try-with-resources  
//        try (FileInputStream fis = new FileInputStream("db.properties")) {  
//            prop.load(fis);  
//        } catch (Exception e) {  
//            System.out.println("Error loading properties file: " + e.getMessage());  
//            return; // Exit if properties not loaded  
//        }  
//  
//        // Step 3: Get database credentials from file  
//        String url = prop.getProperty("dburl");  
//        String user = prop.getProperty("user");  
//        String password = prop.getProperty("password");  
//  
//        // Step 4: Connect to database and fetch employee details  
//        try (Connection conn = DriverManager.getConnection(url, user, password);  
//             Statement stmt = conn.createStatement();  
//             ResultSet rs = stmt.executeQuery("SELECT * FROM employees")) {  
//  
//            System.out.println("Connection Successful!");  
//            System.out.println("Employees Data:");  
//            System.out.println("*****");  
//  
//            // Step 5: Display data  
//            while (rs.next()) {  
//                String fname = rs.getString("firstName");  
//                String empNo = rs.getString("employeeNumber");  
//                String jtitle = rs.getString("jobTitle");  
//  
//                System.out.println("Employee No: " + empNo);  
//                System.out.println("First Name : " + fname);  
//                System.out.println(empNo + ", " + fname + ", " + jtitle);  
//            }  
//        }  
//    }  
}
```

```
//  
// } catch (Exception e) {  
//     System.out.println("Connection Failed: " + e.getMessage());  
// }  
// }  
  
//import com.mysql.cj.jdbc.MysqlDataSource;  
//import java.io.FileInputStream;  
//import java.sql.Connection;  
//import java.util.Properties;  
//  
//public class Main {  
//    public static void main(String[] args) {  
//  
//        Properties prop = new Properties(); // ✅ Move this outside the try block  
//  
//        // Step 1: Load db.properties file  
//        try (FileInputStream fis = new FileInputStream("db.properties")) {  
//            prop.load(fis);  
//        } catch (Exception e) {  
//            System.out.println("Error loading properties file: " + e.getMessage());  
//            return; // Exit if file not loaded  
//        }  
//  
//        // Step 2: Read values from the properties file  
//        String url = prop.getProperty("dburl");  
//        String user = prop.getProperty("user");  
//        String password = prop.getProperty("password");  
//  
//        // Step 3: Configure the MySQL DataSource  
//        MysqDataSource dataSource = new MysqDataSource();  
//        dataSource.setURL(url);  
//        dataSource.setUser(user);  
//        dataSource.setPassword(password);  
//  
//        // Step 4: Connect to the database
```

```
// try (Connection con = dataSource.getConnection()) {
//     System.out.println("Connected to database successfully!");
// } catch (Exception e) {
//     System.out.println("Connection Failed: " + e.getMessage());
// }
// }

//we need not to be close datasource it automatically done
//import com.mysql.cj.jdbc.MysqlDataSource;
//import java.io.FileInputStream;
//import java.sql.Connection;
//import java.sql.ResultSet;
//import java.sql.Statement;
//import java.util.Properties;
//
//public class Main {
//    public static void main(String[] args) {
//
//        // Step 1: Load properties file
//        Properties prop = new Properties();
//
//        try (FileInputStream fis = new FileInputStream("db.properties")) {
//            prop.load(fis);
//        } catch (Exception e) {
//            System.out.println("Error loading properties file: " + e.getMessage());
//            return;
//        }
//
//        // Step 2: Read database details
//        String url = prop.getProperty("dburl");
//        String user = prop.getProperty("user");
//        String password = prop.getProperty("password");
//
//        // Step 3: Create a MySQLDataSource object
//        MysqlDataSource dataSource = new MysqlDataSource();
//        dataSource.setURL(url);
//        dataSource.setUser(user);
//        dataSource.setPassword(password);
//
//        // System.out.println("Connecting to the database...");
//    }
//}
```

```

//    // Step 4: Establish connection and query data
//    try (Connection conn = dataSource.getConnection();
//        Statement stmt = conn.createStatement();
//        /**
//         * Statement is used to send SQL queries to the database.
//        */
//        ResultSet rs = stmt.executeQuery("SELECT firstName, employeeNumber,
//        jobTitle FROM employees")) {
//
//        /**
//         * System.out.println("Connection Successful!");
//         * System.out.println("*****");
//         */
//
//        while (rs.next()) {
//            String fname = rs.getString("firstName");
//            String empNo = rs.getString("employeeNumber");
//            String jtitle = rs.getString("jobTitle");
//
//            /**
//             * System.out.println(empNo + ", " + fname + ", " + jtitle);
//            */
//        }
//
//        /**
//         * } catch (Exception e) {
//         *     System.out.println("Connection Failed: " + e.getMessage());
//         */
//    }
//}

//
```

```

//import com.mysql.cj.jdbc.MysqlDataSource;
//
//import java.io.FileInputStream;
//import java.sql.Connection;
//import java.sql.DriverManager;
//import java.sql.ResultSet;
//import java.sql.Statement;
//import java.util.Properties;
//
//public class Main {
//    public static void main(String[] args) {
//        Properties prop = new Properties();
//
//        /**
//         * // Step 2: Load db.properties file safely using try-with-resources
//         */
//        try (FileInputStream fis = new FileInputStream("db.properties")) {

```

```
//      prop.load(fis);
//  } catch (Exception e) {
//      System.out.println("Error loading properties file: " + e.getMessage());
//      // Exit if properties not loaded
//  }
//
//  // Step 3: Get database credentials from file
//  String url = prop.getProperty("dburl");
//  String user = prop.getProperty("user");
//  String password = prop.getProperty("password");
//  String query = "SELECT * FROM employees";
//
//  MysqlDataSource datasource = new MysqlDataSource();
//  datasource.setURL(url);
//  datasource.setUser(user);
//  datasource.setPassword(password);
//  // test this connection
//  try(Connection conn = datasource.getConnection();
//      Statement stmt = conn.createStatement();
//      ResultSet rs = stmt.executeQuery(query)) {
//
//      System.out.println("Connected to database successfully!");
//      System.out.println("Employee Number | Name | job title");
//      //
//      while(rs.next()) {
//      //
//          String empNo= rs.getString("employeeNumber");
//          String fname= rs.getString("firstname");
//          String jTitle= rs.getString("jobTitle");
//          System.out.printf("%s | %s | %s \n", empNo, fname, jTitle);
//      }
//  } catch(Exception e){
//      System.out.println(e.getMessage());
//  }
//
// }
//}
```

```
//import java.io.FileInputStream;
```

```
//import java.sql.Connection;
//import java.sql.DriverManager;
//import java.sql.Statement;
//import java.util.Properties;
//
//public class Main
//{
//    public static void main(String[] args) {
//        Properties prop = new Properties();
//
//        // Step 1: Load db.properties file
//        try (FileInputStream fis = new FileInputStream("db.properties")) {
//            prop.load(fis);
//        } catch (Exception e) {
//            System.out.println("Error loading properties file: " + e.getMessage());
//            return; // stop execution if properties not loaded
//        }
//
//        // Step 2: Read database credentials
//        String url = prop.getProperty("dburl");
//        String user = prop.getProperty("user");
//        String password = prop.getProperty("password");
//
//        // Step 3: Create SQL query to insert data
//        String query = "INSERT INTO employees (employeeNumber, firstName, lastName, extension, email, officeCode, reportsTo, jobTitle) " +
//                      "VALUES (2208, 'nandhini', 'srinivasan', 'x2312', 'amol@classicmodelcars.com',
//                      4, 1102, 'Sales Rep')";
//
//        // Step 4: Connect to database and execute query
//        try (Connection conn = DriverManager.getConnection(url, user, password);
//             Statement stmt = conn.createStatement()) {
//
//            int rows = stmt.executeUpdate(query);
//            System.out.println(rows + " record(s) inserted successfully!");
//
//        } catch (Exception e) {
//            System.out.println("Database error: " + e.getMessage());
//        }
//    }
//}
```

```

// Defines the package name for organizing your Java files

import java.io.FileInputStream; // To read the db.properties file
import java.sql.Connection; // Represents a connection to the database
import java.sql.DriverManager; // Used to get database connections
import java.sql.Statement; // Used to execute SQL statements
import java.util.Properties; // To load key-value pairs from a properties file

public class Main {
    public static void main(String[] args) {

        // Step 1: Create a Properties object to hold database connection info
        Properties prop = new Properties();

        // Step 2: Load db.properties file using try-with-resources
        // try-with-resources automatically closes the FileInputStream after use
        try (FileInputStream fis = new FileInputStream("db.properties")) {
            // Load the properties (key=value pairs) from the file
            prop.load(fis);
        } catch (Exception e) {
            // If there's an error (e.g., file not found), print error message
            System.out.println("Error loading properties file: " + e.getMessage());
            return; // Exit the program if properties file not loaded properly
        }

        // Step 3: Read database connection details from the properties file
        // The file should contain keys like dburl, user, and password
        String url = prop.getProperty("dburl"); // e.g.
        jdbc:mysql://localhost:3306/classicmodels
        String user = prop.getProperty("user"); // e.g. root
        String password = prop.getProperty("password"); // e.g. 1234

        // Step 4: Write the SQL INSERT query to add a new employee record
        String query = "UPDATE employees SET jobTitle='Junior Developer' WHERE
employeeNumber=2208";

        // Step 5: Establish the connection and execute the SQL statement
        // try-with-resources ensures the connection and statement are closed
        automatically
        try (Connection conn = DriverManager.getConnection(url, user, password); // Open

```

```
connection

    Statement stmt = conn.createStatement() { // Create statement object to send
SQL commands

    // Step 6: Execute the INSERT query
    // executeUpdate() is used for INSERT, UPDATE, or DELETE queries
    int rows = stmt.executeUpdate(query);

    // Step 7: Print confirmation of how many records were inserted
    System.out.println(rows + " record(s) updated successfully!");

} catch (Exception e) {
    // If any SQL or connection error occurs, print the error message
    System.out.println("Database error: " + e.getMessage());
}

}
```